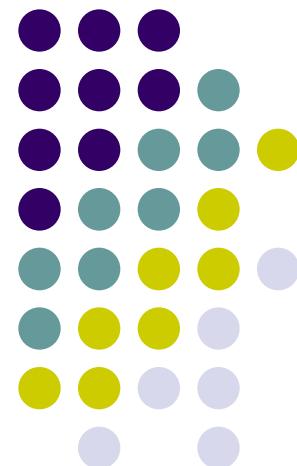
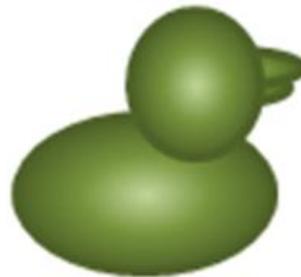


ALGORITMOS Y ESTRUCTURAS DE DATOS 1

Práctica: CUACKER
Sesión 3



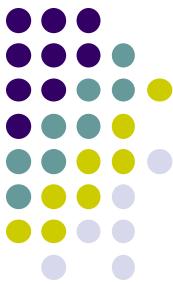


Programación modular

Videotutorial 7

- Cuando el programa crece mucho, es necesario descomponerlo en partes: **módulos**.
 - Descomposición modular del programa.
 - Ficheros de cabecera y de implementación.
 - Compilación separada.
 - Compilación con make.
- Descomposición modular: agrupar funciones relacionadas.
- Reparto equilibrado y lógico. Los módulos no deben ser ni muy pequeños, ni muy grandes.

Programación modular



- Para cada módulo, tendremos dos ficheros: de cabecera (.h o .hpp) y de implementación (.cpp)
 - Fichero de cabecera:** la parte pública, el interfaz.
 - Fichero de implementación:** código de las operaciones.

modulo.h

```
// Modulo uno //
#ifndef _MODULO_UNO
#define _MODULO_UNO

void funcion();

extern int errorCode;

class Persona {
    ...
};

#endif
```

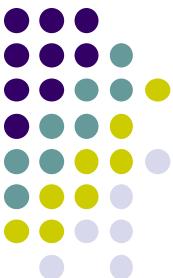
modulo.cpp

```
#include "modulo.h"

void funcion() {
    ...
}

int errorCode= 0;

void Persona::escribir() {
    ...
}
```



Programación modular

- ¿Qué contiene el fichero de cabecera?
 - Directivas de **compilación condicional**: #ifndef ... #endif para evitar inclusiones recursivas o repetidas.
 - Declaración de las **funciones públicas**.
 - Declaración **extern** de las posibles variables públicas.
 - Declaración de los **tipos de datos**, por ejemplo, enumerados o clases.
 - También puede tener **#includes** de otros módulos que requieran los usuarios del presente módulo.

modulo.h

```
// Modulo uno //
#ifndef _MODULO_UNO
#define _MODULO_UNO

void funcion ();

extern int errorCode;

class Persona {
    ...
};

#endif
```



Programación modular

- ¿Qué contiene el fichero de implementación?
 - Normalmente empezará con un **#include** de su propio fichero de cabecera. Pueden incluirse otros módulos, si son necesarios.
 - Definición de las **variables públicas** declaradas extern.
 - Implementación de las **funciones públicas**.
 - Implementación de los **métodos de las clases**, con clase::método.
 - Puede contener otras operaciones (o variables) de **uso interno**.

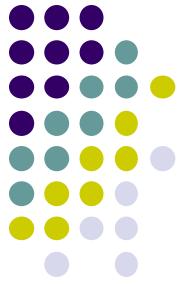
```
modulo.cpp
#include "modulo.h"

int errorCode= 0;

void auxiliar (int n) {
    ...
}

void funcion () {
    ...
}

void Persona::escribir() {
    ...
}
```



Compilación separada

- Supongamos un proyecto con tres módulos:
modulo1(.h,.cpp), modulo2(.h,.cpp), modulo3.cpp
- **Compilación conjunta:**
>> g++ modulo1.cpp modulo2.cpp modulo3.cpp
- **Compilación separada:** compilar cada módulo por separado (a código objeto) y luego lincar (enlazar).
>> g++ -c modulo1.cpp
>> g++ -c modulo2.cpp
>> g++ -c modulo3.cpp
>> g++ modulo1.o modulo2.o modulo3.o

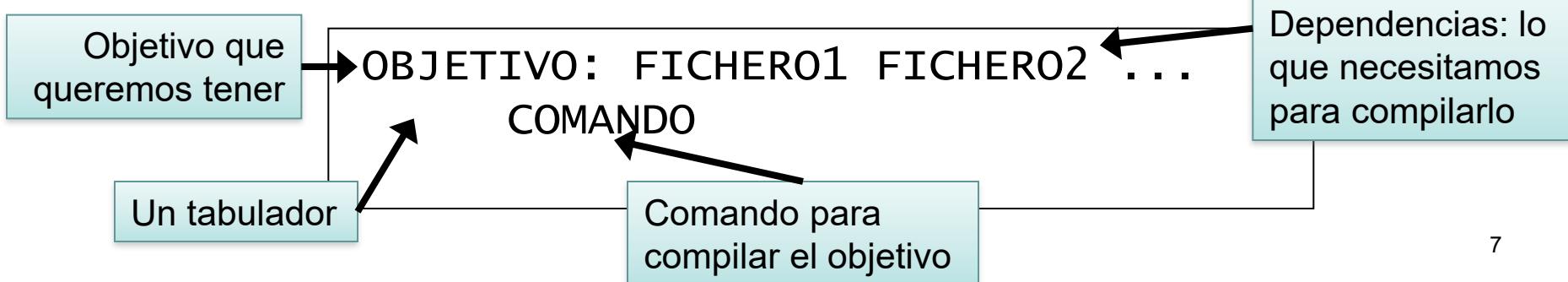


Compilación con make

- Ventajas: solo se necesita recompilar lo que se haya modificado durante el desarrollo.
- Pero, ¿cómo compilar el proyecto completo?
- El comando **make** de Linux permite automatizar el proceso de compilación separada.

>> make

- Busca un fichero de texto llamado **Makefile** donde debemos describir cómo compilar el proyecto.





Compilación con make

- El objetivo que aparezca en primer lugar dentro del Makefile se considera el objetivo principal.

Makefile

```
# Un comentario de make
a.out: modulo1.o modulo2.o modulo3.o
        g++ modulo1.o modulo2.o modulo3.o

modulo1.o: modulo1.cpp modulo1.h
        g++ -c modulo1.cpp

modulo2.o: modulo2.cpp modulo2.h modulo1.h
        g++ -c modulo2.cpp

modulo3.o: modulo3.cpp modulo1.h modulo2.h
        g++ -c modulo3.cpp
```



Compilación con make

- Ahora se compila simplemente con make:

```
>> make
```

- Nuestro programa se compone de muchos ficheros. ¿Cómo se envía a Mooshak?

- El ejecutable generado se debe llamar a.out
- Empaquetamos todos los .cpp, .h, .hpp y Makefile, dentro de un tar sin comprimir:

```
>> tar -cf archivo.tar *.cpp *.h Makefile
```

- Mooshak: (1) tomará el fichero TAR; (2) descomprime los archivos *.cpp, *.h, *.c, *.hpp y Makefile; (3) ejecuta ">> make"; y (4) ejecuta ">> ./a.out".



Semana 3: ejercicios 005 y 006

Planificación práctica



005 – Intérprete de comandos

- Construir un intérprete de comandos que reconozca los comandos del problema.
- Objetivo: crear el **esqueleto** del programa principal de la práctica.
- Cuestiones a tratar:
 - Crear un procedimiento para cada comando.
 - Crear un intérprete que haga un análisis de casos.
 - Según el caso, se llama a un procedimiento.



005 – Intérprete de comandos

- Ejemplo entrada/salida:

mcuac RafaelNaval

25/10/2011 13:45:11

¡Feliz Navidad #amigosdenaval!

pcuac RafaelNaval

28/11/2011 11:27:08

5

last 5

follow Perico

mcuac GinesGM

6/5/2012 16:00:00

Dicen en #eltiempo que este lunes...

date 28/11/2011 11:27:04 28/11/2012 11:27:08

pcuac Gutierrez

1/1/2013 00:00:00

27

last 100

pcuac GinesGM

1/1/2013 00:00:01

30

tag #eltiempo

1 cuac

2 cuac

last 5

1. RafaelNaval 28/11/2011 11:27:08
Enhorabuena, campeones!

Total: 1 cuac

follow Perico

1. RafaelNaval 28/11/2011 11:27:08
Enhorabuena, campeones!

Total: 1 cuac

3 cuac

date 28/11/2011 11:27:04 28/11/2012 11:27:08
1. GinesGM 6/5/2012 16:00:00
Dicen en #eltiempo que este lunes...

Total: 1 cuac

4 cuac

last 100

1. Gutierrez 1/1/2013 00:00:00
Me despido hasta la proxima. Buen viaje!

Total: 1 cuac

5 cuac

tag #eltiempo

1. GinesGM 1/1/2013 00:00:01

El que quiera saber mas, que se vaya a Salamanca.

Total: 1 cuac



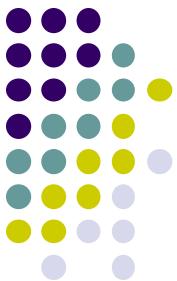
005 – Intérprete de comandos

- Más adelante daremos contenido a los comandos. Por ahora estarán todos “vacíos”.
- Usamos un **contador** del número de cuacs y el último cuac.
- El **main** queda parecido a lo siguiente:

```
int contador = 0;  
Cuac actual;  
...  
int main (void) {  
    string comando;  
    while (cin >> comando && comando!="exit")  
        Interprete(comando);  
}
```

¡Simple y
claro!

005 – Intérprete de comandos



- Análisis de casos del intérprete:

- El intérprete de comandos será un procedimiento que recibe el comando.
- Analiza el caso y llama a la operación correspondiente.

```
void Interprete (string comando)
{
    if (comando=="pcuac") procesar_pcuac();
    else if (comando=="mcuac") procesar_pcuac();
    else if (comando=="last") procesar_last();
    ...
}
```

006 – Diccionario de cuacs con listas

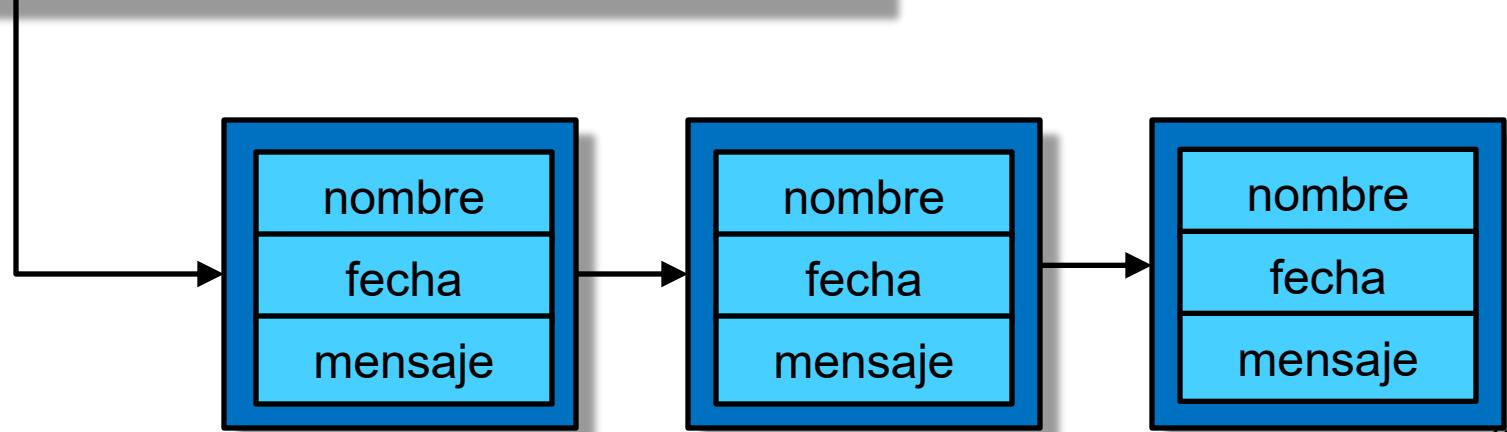
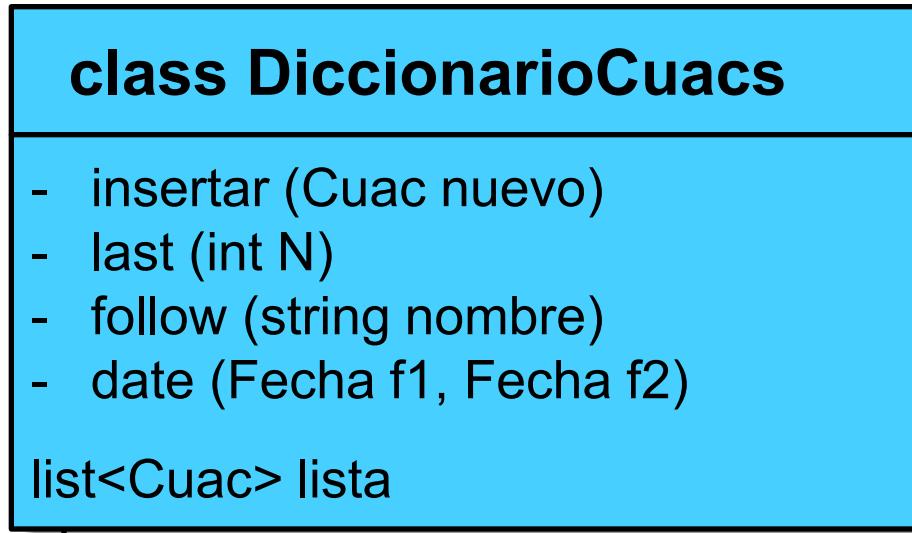


- Añadir una clase **diccionario de cuacs**.
- Tendrá una implementación sencilla con listas de Cuac: el tipo **list<T>** de las STL de C++.
- ¡No necesitamos tocar el intérprete de comandos!
Solo las operaciones básicas del mismo:
 - **procesar_mcuac** y **procesar_pcuac**: llaman al método **insertar** del diccionario.
 - **procesar_last**: llama al método **last** del diccionario.
 - **procesar_follow**: llama al método **follow** del dic.
 - **procesar_date**: llama al método **date** del diccionario.

006 – Diccionario de cuacs con listas



- El diccionario de cuacs.



006 – Diccionario de cuacs con listas

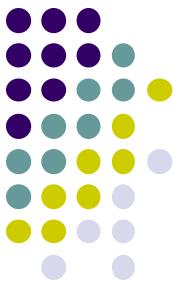


```
class DiccionarioCuacs {  
    private:  
        list<Cuac> lista;  
        int contador;  
    public:  
        DiccionarioCuacs ();  
        void insertar (Cuac nuevo);  
        void last (int N);  
        void follow (string nombre);  
        void date (Fecha f1, Fecha f2);  
        int numElem ()  
            {return contador;}  
};
```

Ojo: Como no se usa memoria dinámica (no aparecen punteros) no se necesita reservar memoria ni definir un destructor.

Para simplificar suponemos que las operaciones escriben directamente el resultado.

006 – Diccionario de cuacs con listas



- El conjunto será un objeto de esa clase:

```
DiccionarioCuacs dic;
```

- ¿Dónde definir este objeto?

- Como una variable global a la que tiene acceso el intérprete.
 - O como una variable local del main. En este caso, pasarla por referencia a todas las funciones.
 - O bien redefinimos el intérprete de comandos como una nueva clase...
- Por simplicidad, suponemos que es una variable global. Aunque no tiene por qué ser lo mejor...

¡Aquí tenéis que empezar a tomar vuestras propias decisiones de diseño!

006 – Diccionario de cuacs con listas



- Las operaciones del intérprete de comandos se basan en las del diccionario:

```
void procesar_pcuac ()  
{  
    Cuac nuevo;  
    nuevo.leer_pcuac();  
    dic.insertar(nuevo);  
    cout << dic.numElem() << " cuac";  
}  
void procesar_follow ()  
{  
    string nombre;  
    cout << "follow " << nombre << endl;  
    dic.follow(nombre);  
}  
...
```

006 – Diccionario de cuacs con listas



- Y ahora implementar los métodos de las clases:

```
DiccionarioCuacs::DiccionarioCuacs ()  
{  
    contador= 0;  
}  
void DiccionarioCuacs::insertar (Cuac nuevo) {  
    Insertar el nuevo cuac en lista de forma ordenada  
    contador++  
}  
...
```

¡Y acordarse de modularizar el programa!



Breve introducción al tipo list<T>

- **STL: Standard Template Library.** Biblioteca estándar de C++, que contiene contenedores genéricos: list<T>, vector<T>, queue<T>, set<T>, map<C,V>, etc.
- Muy interesante, para aprovechar las implementaciones de los tipos lista, cola, etc.
 - **Brevísimo manual de introducción a las listas de STL:**
<http://dis.um.es/~ginesgm/files/doc/aed/listSTL.html>
 - **Referencia completa de las STL (sección list):**
<http://www.cplusplus.com/reference/stl/list/>
 - **Manual de las STL para principiantes de la OIE:**
<http://dis.um.es/~ginesgm/files/doc/aed1/guiastl.pdf>



Breve introducción al tipo list<T>

- **list<T>: listas genéricas de tipo T.**

```
#include <list>
using namespace std;
```

- **Instanciación.** Al definir una variable de tipo lista, hay que sustituir T por el tipo que se quiere almacenar.
- **Listas creadas estáticamente:**

```
list<int> lista1;
list<string> lista2;
```

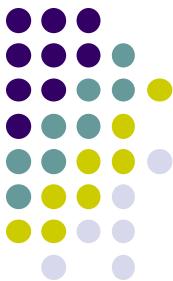
- **Listas creadas dinámicamente:**

```
list<int> *plista= new list<int>;
```

```
...
```

```
delete plista;
```

- En ambos casos, las listas se inicializan por defecto a listas vacías.



Breve introducción al tipo `list<T>`

- **Inserción en una lista:**

- Insertar por el principio: **`push_front`**.

```
lista1.push_front(14);  
lista2.push_front("Hola");  
plista->push_front(9);
```

- Insertar por el final: **`push_back`**.

```
lista1.push_back(3);  
plista->push_back(7);
```

- Tamaño de una lista: **`size`**.

```
int tam= lista1.size();
```

¡Cuidado! Según el estándar, puede ser un $O(n)$.

- Borrar todos los elementos: **`clear`**.

```
lista1.clear();  
plista->clear();
```

Breve introducción al tipo `list<T>`



- Para recorrer una lista se usa un tipo **iterador**.

```
list<int> lista;  
list<int>::iterator itLista;
```

- Operaciones principales sobre los iteradores:**

- Iniciar** al principio:

```
itLista= lista.begin();
```

- Avanzar** el iterador (o retroceder):

```
itLista++;    itLista--;
```

- Consultar el **elemento actual** apuntado por el iterador.

```
int actual= *itLista;
```

- Consultar si se ha llegado al **final de la lista**:

```
if (itLista==lista.end()) ...
```

- Insertar en una lista en una posición dada: **insert**.

```
lista.insert(itLista, valor);
```

- Eliminar el valor apuntado por el iterador: **erase**.

```
lista.erase(itLista);
```

La posición `lista.end()` es *pasado* el último elemento. Es decir, cae fuera de la lista

Insertar justo *delante* de la posición apuntada por el iterador.

Breve introducción al tipo `list<T>`



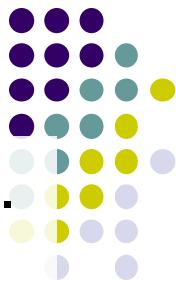
- El tratamiento secuencial de elementos con iteradores en C++ sigue el Primer Modelo de Acceso secuencial:

Secuencias	Iteradores
Comenzar	<code>it = lista.begin()</code>
EA	<code>*it</code>
Avanzar	<code>it++</code>
EA = MarcaFin	<code>it == lista.end()</code>

Nota de
I.P.

- ¡OJO! El acceso al elemento actual cuando se ha llegado a la marca fin (es decir, el acceso a `*it` cuando se cumple que `it == lista.end()`) produciría un error. [Nota de Paco Montoya](#)
- En las expresiones booleanas que combinan ambas cosas deben utilizarse los operadores booleanos en *cortocircuito* (`&&` y `||`) para evitar accesos erróneos.

Breve introducción al tipo list<T>



- **Ejemplo 1.** Recorrer una lista de cadenas y escribirlas.

```
list<string> lista;  
lista.push_front("Hola");  
...  
list<string>::iterator it;  
for (it= lista.begin(); it!=lista.end(); it++)  
    cout << "Valor: " << *it << endl;
```

- **Ejemplo 2.** Inserción ordenada de cadena en una lista (sin insertarla si ya estaba).

```
list<string>::iterator it= lista.begin();  
while (it!=lista.end() && *it<cadena)  
    it++;  
if (it==lista.end() || *it!=cadena)  
    lista.insert(it, cadena);
```

Nota: devolver un puntero al objeto de la lista: &(*it)