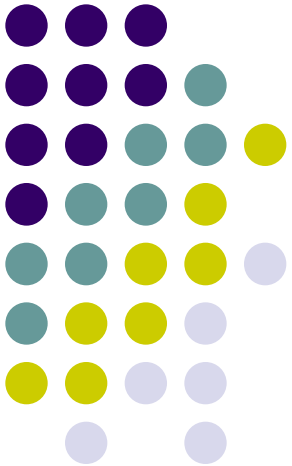
http://webs.um.es/frgarcia/teaching.htm

ALGORITMOS Y ESTRUCTURAS DE DATOS

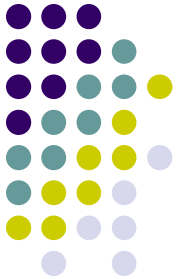
PRÁCTICA. TEMAS 2 Y 3

SESIÓN 4



Francisco García Sánchez
frgarcia@um.es
Despacho 2.31

INDICE DE CONTENIDOS



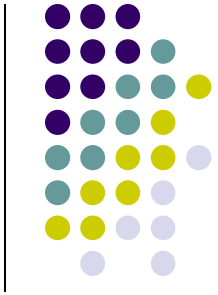
- Planificación práctica (semana 4)
  - Ejercicio 201
  - Ejercicio 202
  - Ejercicio 203
- Introducción a la STL en C++
  - Conjuntos
  - Algunas funciones y algoritmos de interés
- Programación modular

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 201
  - Tablas de dispersión de palabras
    - Añadir estructuras de datos más eficientes
    - Combinación de estructuras que nos permitan resolver de forma eficiente cada una de las consultas de nuestro sistema
      - **Crear tablas de dispersión de palabras donde para cada palabra se tendrá su número de apariciones**
      - Tipo de dato **`Palabra`** con la palabra y el número de apariciones

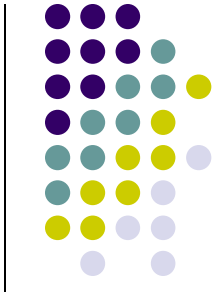
Planificación práctica



Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 201
  - Tablas de dispersión de palabras
    - Para conseguir que las búsquedas sean independientes de mayúsculas/minúsculas, todas las palabras (tanto del nombre como de la descripción) deben **convertirse usando la función definida en el ejercicio 003**
    - La estructura diseñada de tablas de dispersión debe **integrarse** junto con la colección de productos creada en los ejercicios anteriores

Planificación práctica



Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 201
  - Tablas de dispersión de palabras
    - En este ejercicio solo aparecerán las consultas por **palabras**.
    - Las consultas serán siempre de *una sola palabra* y *el **resultado** será únicamente el número de productos en los que aparece dicha palabra* (ya sea en el nombre o en la descripción, de forma indiferente).

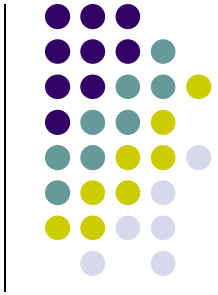
Comando **`insertar`** → `N productos`
Comando **`palabras`** → `Total: X productos`

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 202
  - Búsqueda simple de palabras
    - Completar la operación **`palabras`** con consultas de una sola palabra.
    - Modificar el tipo de dato **`Palabra`** añadiendo un conjunto de referencias a los productos (*y las operaciones para insertar un nuevo producto y para listarlos de forma ordenada*)

Planificación práctica



Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- `202`: *recomendado* `set<T>` *de las STL*
  1. No se deben almacenar productos sino referencias a los productos, ya que no deben duplicarse los productos
  2. Los productos deben listarse de forma ordenada, por lo que es conveniente es utilizar una estructura donde los productos estén ordenados (inserción y listado rápidos)
  3. En un producto puede repetirse muchas veces una palabra. Es adecuado que se tenga en cuenta para no repetir las inserciones

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 202
  - Búsqueda simple de palabras
    - Salida del comando **`palabras`**:
      - Escribir los productos (según el formato de salida indicado en el ejercicio 004), precedidos por un número consecutivo (empezando por 1), con un punto y un espacio
      - Los productos estarán ordenados por identificador, de menor a mayor
      - En la última línea se escribirá: "`Total: X productos`", siendo X el número de productos resultantes de esta consulta

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

- 203
  - Búsqueda de múltiples palabras
    - Completar la operación **`palabras`** con consultas de un número indeterminado de palabras.
    - El resultado de la búsqueda múltiple serán los productos que aparezcan en todas las consultas individuales, es decir, la **intersección** de los conjuntos de productos de las palabras consultadas.

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

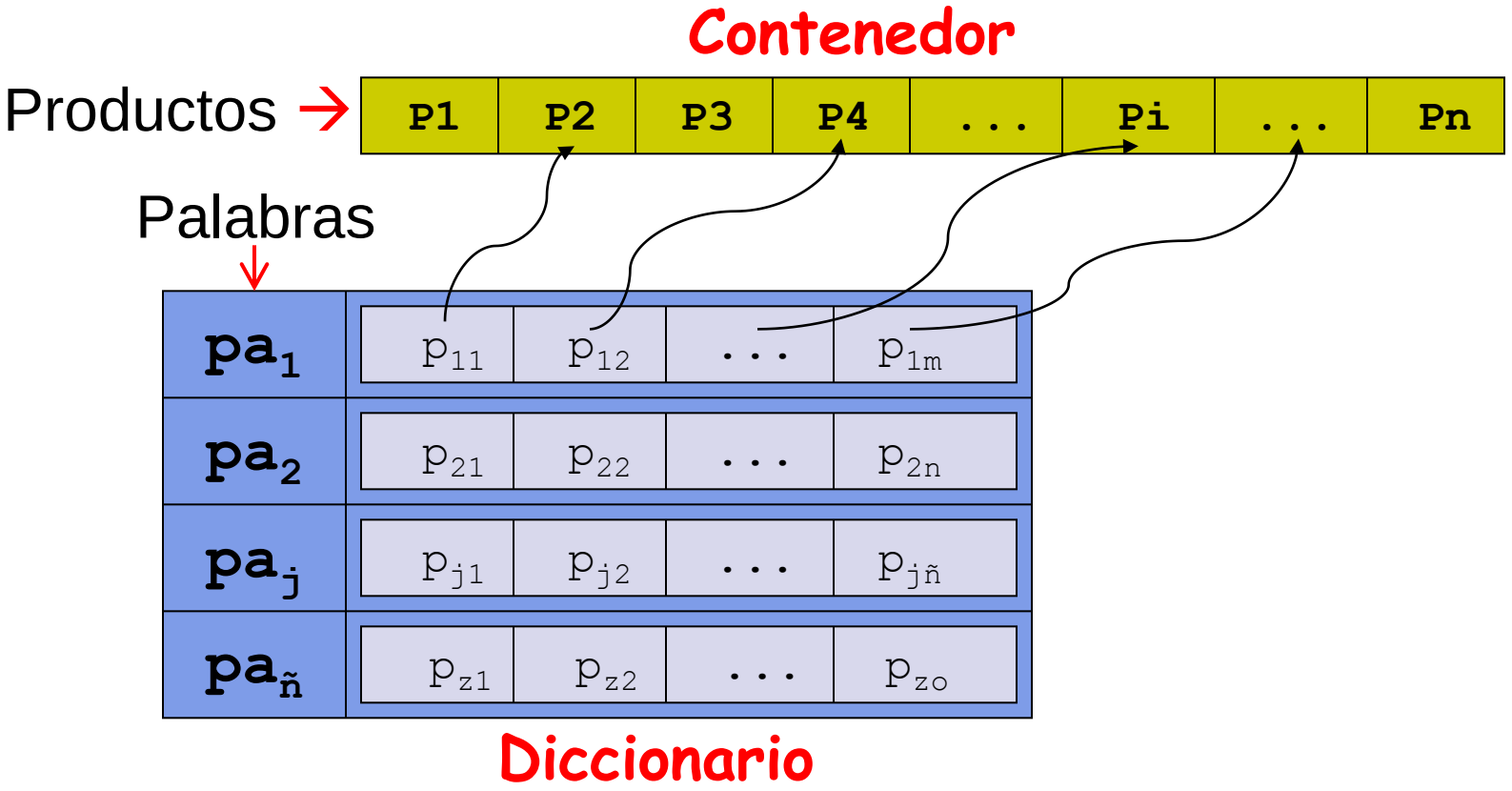
- 203
  - Búsqueda de múltiples palabras
    - La principal cuestión de este ejercicio es cómo conseguir eficiencia en la intersección de las consultas individuales.
    - En clase se vio cómo la intersección de conjuntos se puede conseguir de forma eficiente usando listas ordenadas.
    - Esto es aplicable en nuestro caso.

Planificación práctica

Semana 4. 12 de noviembre: tablas de dispersión de cadenas (201), tablas de dispersión para la operación palabras (202) y búsqueda de múltiples palabras (203).

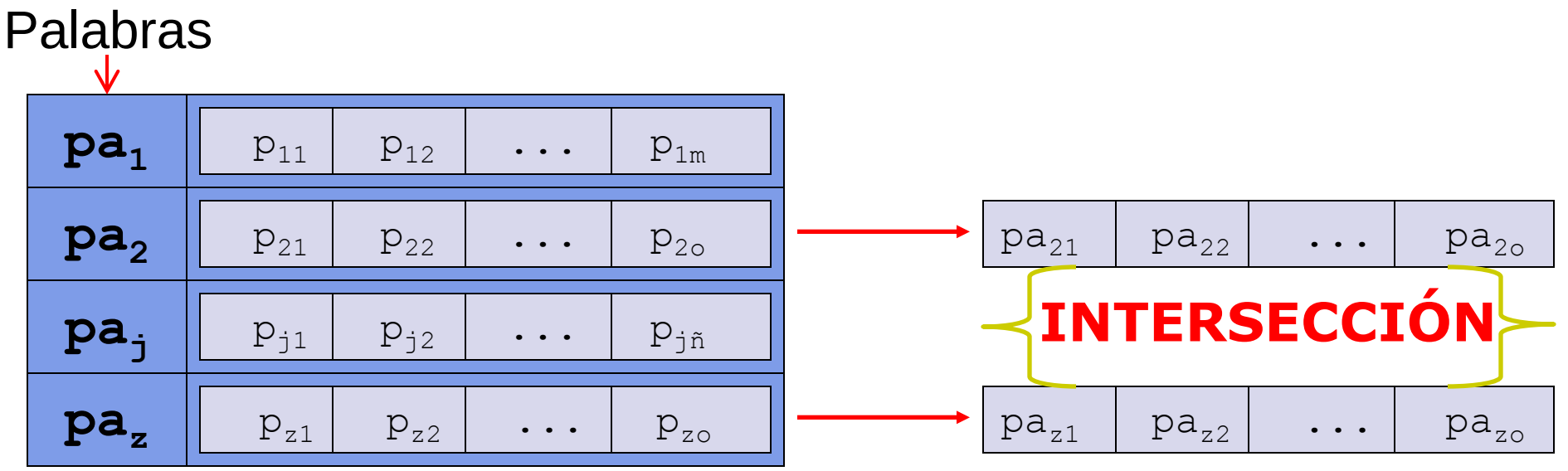
- 203
  - Búsqueda de múltiples palabras
    - **Entrada**: el comando **`palabras`** puede contener muchas palabras
    - **Salida**: mismo formato que para el ejercicio 202

Planificación práctica

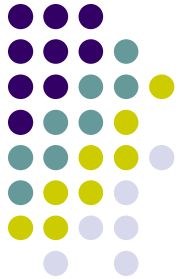


Planificación práctica

- *Búsqueda:* `palabras` pa_2 pa_z



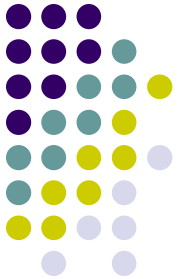
Introducción a la STL en C++



- Conjuntos: `set<K>`
  - **Conjunto**: estructura donde podemos guardar colecciones de objetos o elementos no repetidos.
    - `set<K>`

http://www.cplusplus.com/reference/stl/set/

Introducción a la STL en C++

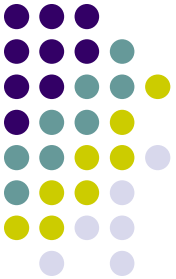


- Conjuntos: `set<K>`
  - `#include <set>`
  - Operaciones básicas (`set<K> s`)
    - **`s.insert(c)`**: insertar una clave `c`
    - **`s.erase(c)`**: elimina `c` del conjunto
    - **`s.size()`**: indica tamaño del conjunto
    - **`s.empty()`**: `true` si tamaño es 0
    - **`s.clear()`**: elimina todos los elementos
    - **`s.find(c)`**: busca c en el conjunto; devuelve un iterador

Introducción a la STL en C++

- Conjuntos: `set<K>`
  - Manipulación con iteradores: `set<K>::iterator`
    - **`it = s.begin()`**: iterador a inicio del conjunto
    - **`it = s.end()`**: iterador a final del conjunto
    - **`*it`**: para encontrar el elemento apuntado por el iterador `it`

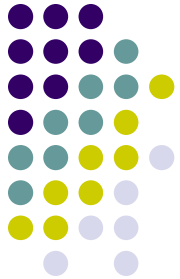
Introducción a la STL en C++



- Conjuntos y diccionarios: `set<K>` y `map<K,T>`

```
set<int> s;
s.insert(1);
set<int>::iterator it = s.begin();
while(it != s.end()) {
   int k1 = (*it)*3;
   if (k1<100000) s.insert(k1);
   int k2 = (*it)*5;
   if (k2<100000) s.insert(k2);
   ++it;
}
```

http://www.cplusplus.com/reference/algorithm/set_union/



Introducción a la STL en C++

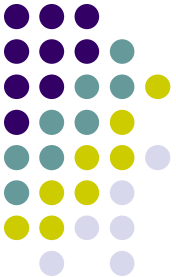
Incluir librería «algorithm»: `#include <algorithm>`

- Algoritmos y funciones interesantes
  - `set_union`: Unión de dos rangos ordenados

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator set_union ( InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2, InputIterator2
last2, OutputIterator result ) {
   while (true) {
       if (*first1<*first2) *result++ = *first1++;
       else if (*first2<*first1) *result++ = *first2++;
       else { *result++ = *first1++; first2++; }
       if (first1==last1) return copy(first2,last2,result);
       if (first2==last2) return copy(first1,last1,result);
   }
}
```

```cpp
// set_union example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);                      // 0  0  0  0  0  0  0  0  0  0
  vector<int>::iterator it;

  sort (first,first+5);     //  5 10 15 20 25
  sort (second,second+5);   // 10 20 30 40 50

  it=set_union (first, first+5, second, second+5, v.begin());
                                          // 5 10 15 20 25 30 40 50  0  0

  cout << "union has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

http://www.cplusplus.com/reference/algorithm/set_intersection/



Introducción a la STL en C++

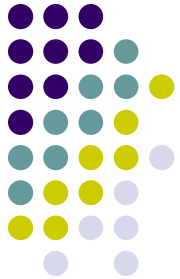
Incluir librería «algorithm»: `#include <algorithm>`

- Algoritmos y funciones interesantes
  - `set_intersection`: intersección

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator set_intersection ( InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2, InputIterator2
last2, OutputIterator result ) {
   while (first1!=last1 && first2!=last2) {
      if (*first1<*first2) ++first1;
      else if (*first2<*first1) ++first2;
      else {
         *result++ = *first1++;
         first2++;
      }
   }
return result; }
```

```cpp
// set_intersection example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);                                  // 0  0  0  0  0  0  0  0  0  0
  vector<int>::iterator it;

  sort (first,first+5);     //  5 10 15 20 25
  sort (second,second+5);   // 10 20 30 40 50

  it=set_intersection (first, first+5, second, second+5, v.begin());
                                                      // 10 20 0  0  0  0  0  0  0  0

  cout << "intersection has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

Programación modular



- Permite descomponer la complejidad de una aplicación en distintos trozos
  - Un ***módulo*** o ***paquete*** contiene un conjunto de funcionalidades relacionadas ≈ clases en C++
  - **Programación modular**: 1 módulo → 2 ficheros
    - Fichero de cabecera (header; **.hpp**):
      - Definición de los tipos de datos (clases)
      - Declaración de funciones (cabecera en las clases)
    - Fichero de implementación (**.cpp**):
      - Implementación de las funciones declaradas (y otras auxiliares)

Modularidad

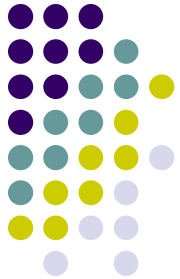
Nuestro programa empieza ya a tener cierto tamaño. Por lo tanto, va siendo necesario aplicar los principios de la **programación modular** para organizar mejor el código. Debemos descomponer el código en diferentes módulos asociados a las distintas funcionalidades que aparecen. Una vez hecha la descomposición modular, ¿cómo enviar el programa al juez?

La solución es usar ficheros TAR. El juez on-line admite como entrada un fichero TAR, que contiene los archivos de código C/C++ necesarios y el `Makefile` para construir el ejecutable. Se deben seguir los siguientes pasos:

- Estando situados dentro del directorio donde se encuentran todos los ficheros de código, ejecutar:
  ```
  >> tar -cf archivo.tar *.cpp *.h Makefile
  ```
- Observar que el parámetro `archivo.tar` indica el nombre del fichero que se quiere generar, y a continuación vienen los archivos a incluir en el TAR. No se deben usar ficheros TAR comprimidos.
- El archivo `Makefile` debe contener las instrucciones adecuadas para compilar el proyecto y generar un **ejecutable llamado** `a.out`. Repetimos: el ejecutable resultante debe llamarse: `a.out`
- El juez on-line: (1) toma el fichero TAR; (2) extrae los archivos `*.cpp`, `*.h`, `*.c`, `*.hpp` y `Makefile`; (3) ejecuta "`>> make`"; y (4) ejecuta "`>> ./a.out`". Si falla alguno de estos pasos, se obtendrá un error.

A partir de este ejercicio y en los sucesivos, se requiere que los programas sean subidos al juez en formato TAR. En los seminarios de repaso de C puedes encontrar una guía abreviada del uso de Makefile.

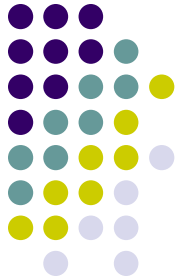
Programación modular (make)



- Pasos en la generación de un programa:
  1. **Compilación**: dado el fichero con el código fuente (`.cpp`) generar un fichero de código objeto (`.o`)
  2. **Linkado o enlace**: dado uno o varios ficheros de código objeto (`.o`) producir un fichero ejecutable
- Por defecto, `g++`, hace los dos pasos sin generar código objeto
  - Utilizar opción `-c` (compilar, no enlazar)

`g++ -c ejemplo.cpp` → Genera `ejemplo.o`

Programación modular (make)



- Compilación en programas de gran tamaño:
  - ☹ Muchos módulos, código fuente largo, mucho tiempo de compilación, modificaciones frecuentes…
  - ☺ Generar fichero objeto de cada módulo por separado; cuando se cambia un módulo, recompilar ese módulo y linkar el programa principal.
  - ☹ Si se modifica fichero cabecera o fuente, otros módulos se pueden ver afectados y hay que recompilarlos…¿cuáles?
  - 👌 Programa **make**: automatizar proceso de compilar

Programación modular (make)

- Programa **make**:
  - Busca y procesa fichero: `makefile`
  - `makefile` indica dependencias entre módulos:

```
OBJETIVO: FICHERO1 … FICHEROP
<Tabulador>COMANDO
```

    - El fichero `OBJETIVO` debe recompilarse cuando se modifiquen (o recompilen) `FICHERO1…FICHEROP`
    - La forma de obtener fichero `OBJETIVO` es ejecutando `COMANDO`

Programación modular (tar)

- Interacción con Mooshak:
  - Mooshak sólo acepta 1 fichero ➔ añadir todos los ficheros involucrados en un único fichero "`.tar`"
  - 201.tar ← pagina.hpp pagina.cpp … buscador.cpp makefile

```
tar cfv 201.tar producto.hpp
 producto.cpp … principal.cpp makefile
```

  - makefile:

```
all:
 g++ -Wall producto.cpp … principal.cpp
 -o a.out
```