# Ejercitación – Preprocesamiento y compilación

Se recomienda resolver los ejercicios en orden. En CLion se encuentran disponibles los siguientes targets:

- ejN, si ($N \in \{1, 2, 3\}$): compila los tests correspondientes al ejercicio N.
- ejN, si ($4 \leq N \leq 8$): compila los tests desde el ejercicio 4 hasta al ejercicio N inclusive.

Los targets también pueden compilarse y ejecutarse sin usar CLion. Para ello:

1. En una consola pararse en el directorio raíz del projecto. En este debería haber un archivo CMakeLists.txt.
2. Ejecutar el comando `$ cmake .` (incluyendo el punto). Esto generará el archivo Makefile.
3. Ejecutar el comando `$ make TARGET` donde TARGET es uno de los targets mencionados anteriormente. Esto creará un ejecutable con el nombre del target en el directorio actual.
4. Ejecutar el comando `$ ./TARGET` siendo TARGET el nombre del target utilizado anteriormente. Esto correrá el ejecutable.

## Ejercicio 1

Extraer la clase `class Periodo` fuera del archivo Fecha.cpp a dos archivos Periodo.h y Periodo.cpp.

## Ejercicio 2

Intentar compilar el target ej2. Si no compila, resolverlo.

## Ejercicio 3

Extraer las declaraciones de tipo `typedef` y de meses (ej.: `const Mes ENERO = 1;`) a un archivo Meses.h.

Luego, extraer las declaraciones de `bool esBisiesto(int)` y `int diasEnMes(int, int)` en un archivo Funciones.h. Extraer las definiciones en Funciones.cpp.

Recordar agregar los `#include` necesarios.

# Ejercitación – Diccionario lineal

La clase `Diccionario` representa un diccionario que asocia claves de tipo `Clave` a valores de tipo `Valor`.

## Ejercicio 4

En el archivo `src/Diccionario.h`, completar la parte privada de la clase `Diccionario`, eligiendo su representación. Se sugiere representar el diccionario como un vector de asociaciones. Una asociación es un par definido de la manera siguiente:

```
struct Asociacion {
  Clave clave;
  Valor valor;
};
```

El vector de asociaciones no debería tener claves repetidas. Si se elige esta representación, la declaración del `struct Asociacion` debe ubicarse en la parte **privada** de la clase `Diccionario`.

En el archivo `src/Diccionario.cpp`, definir los métodos:

- `Diccionario::Diccionario()` – construye un diccionario vacío.
- `void Diccionario::definir(Clave k, Valor v)` – asocia la clave `k` al valor `v`. Si la clave ya existía, sobreescribe su valor.
- `bool Diccionario::def(Clave k) const` – devuelve `true` si y sólo si la clave está definida.
- `Valor Diccionario::obtener(Clave k) const` – devuelve el valor asociado a la clave.

## Ejercicio 5

Agregar un método `borrar` que recibe un clave y la elimina del diccionario. (Si la clave no figuraba en el diccionario, el diccionario no se modifica). Para ello:

- Agregar su declaración en el archivo `src/Diccionario.h`.
- Agregar su implementación en el archivo `src/Diccionario.cpp`.
- En el archivo `test/test_diccionario.cpp` agregar casos de test para probar las siguientes funcionalidades:
  - Borrar una clave existente y verificar que deje de existir.
  - Borrar una clave existente y verificar que las claves restantes sigan existiendo con el mismo valor que tenían.
  - Borrar una clave inexistente y verificar que el diccionario no se haya modificado.

## Ejercicio 6

Agregar un método `std::vector<Clave> Diccionario::claves() const` que devuelve un vector que contiene todas y solamente las claves del diccionario, en algún orden, y sin repetidos. Agregar casos de test para comprobar esta funcionalidad.

**Sugerencia:** para escribir los tests, definir la siguiente función auxiliar:

- `bool esPermutacion(std::vector<Clave> v1, std::vector<Clave> v2)` – devuelve `true` si y sólo si `v1` es una permutación de `v2`.

## Ejercicio 7

Definir el método `bool Diccionario::operator==(Diccionario o)` que devuelve `true` si los diccionarios son iguales. Dos diccionarios son iguales si tienen las mismas claves y el significado asociado a cada clave es el mismo[1]. Agregar casos de test para comprobar esta funcionalidad.

**Importante:** observar que dos diccionarios pueden ser iguales a pesar de que su representación interna sea distinta. Por ejemplo, los dos vectores de asociaciones siguientes representan el mismo diccionario:

| | | |
|---|---|---|
| clave = 1, valor = 10 | clave = 2, valor = 20 | clave = 3, valor = 30 |

| | | |
|---|---|---|
| clave = 2, valor = 20 | clave = 3, valor = 30 | clave = 1, valor = 10 |

## Ejercicio 8

Definir los métodos `operator||` y `operator&&`, con la siguiente especificación:

- `Diccionario Diccionario::operator||(Diccionario o) const` – calcula la "unión" de diccionarios. La unión de dos diccionarios incluye las claves que están en cualquiera de ambos, es decir una clave `k` está definida en el diccionario `d1 || d2` si está definida en `d1` o está definida en `d2`. Además:
  - Si la clave `k` está definida en `d1`, entonces `obtener(d1 || d2, k) == obtener(d1, k)`.
  - Si la clave `k` no está definida en `d1`, entonces `obtener(d1 || d2, k) == obtener(d2, k)`.
- `Diccionario Diccionario::operator&&(Diccionario o) const` – calcula la "intersección" de diccionarios. La intersección de dos diccionarios incluye las claves que están simultáneamente en ambos, es decir una clave `k` está definida en el diccionario `d1 && d2` si está definida en `d1` y está definida en `d2`. Además, `obtener(d1 && d2, k) == obtener(d1, k)`.

Agregar casos de test para comprobar esta funcionalidad.

[1] Todavía no vimos cómo relacionar formalmente la especificación con TADs con la implementación en C++, pero informalmente `d1 == d2` debe ser `true` si y sólo si `d1` y `d2` son observacionalmente iguales.