fjmadrid@uco.es

Estructuras de Datos
Grado de Ing. Informática

Introducción a las prácticas de EEDD

13/02/25

fjmadrid@uco.es

Contenidos

- Evaluación de las prácticas.
- Entorno de desarrollo.
- Repaso de conceptos C++

fjmadrid@uco.es

Evaluación

- 6 prácticas, dos sesiones por práctica.
- Se utiliza la metodología Test Driven Development.
- En cada sesión de prácticas se fijaran unos objetivos y serán evaluados suponiendo un punto extra a sumar a la nota del examen práctico.
- La práctica se evalúa mediante un script de corrección (importante **normativa de entrega**).
- El examen práctico consistirá en resolver algunas partes de las prácticas.

Entorno de desarrollo

- Plataforma de referencia.
- Trabajando con VSCode.
- Trabajando con la línea de comandos.

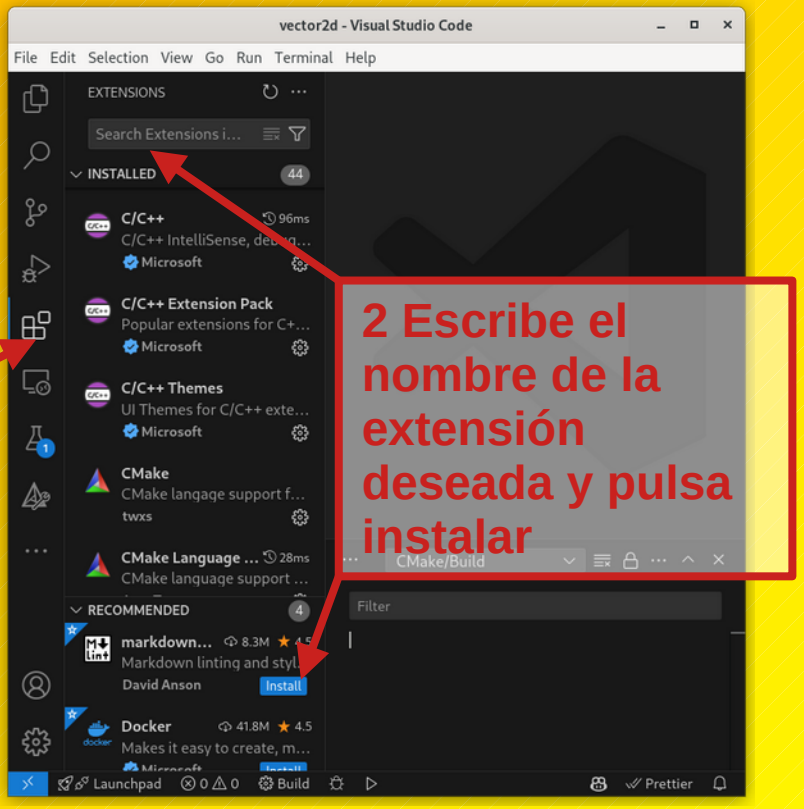
Entorno de desarrollo

- Plataforma de referencia: ThinStation.
- Plataformas soportadas: Linux (debian/ubuntu), Windows (msvc, wsl, cygwin, msys2, ¿mac os?)
- IDE recomendado: VSCode.
- Extensiones VSCode:
  - C++ extension pack
  - Prettier
  - ¿git extension pack?
  - ¿Copilot?

fjmadrid@uco.es

Trabajando con VSCode

- Paso 0: instalar las extensiones necesarias.
  - C++ extension pack
  - Prettier
  - ¿git extension pack?
  - ¿Copilot?

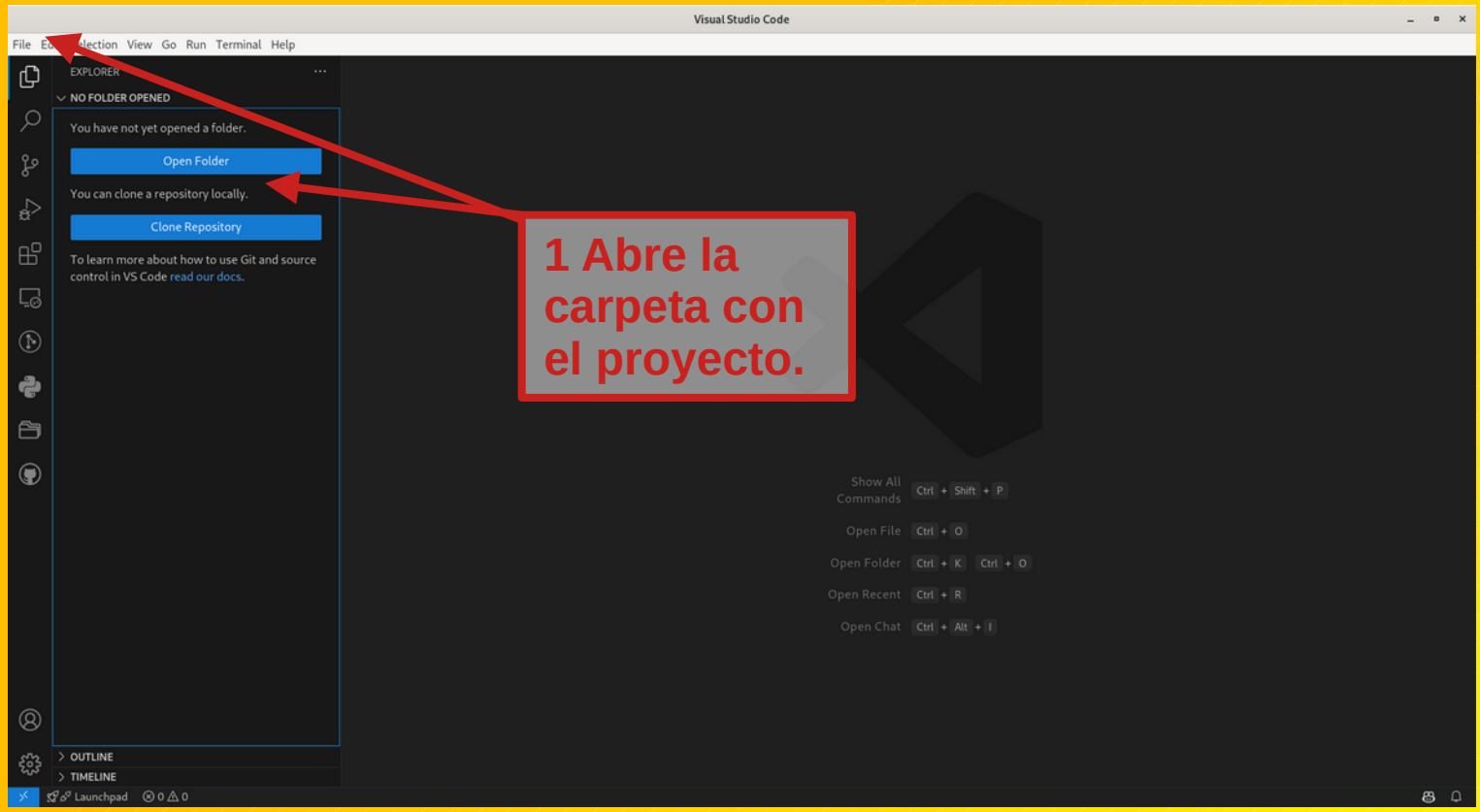


Nota: en ThinStation ya están instaladas las extesiones aunque desactualizas.
Nunca actualices VSCode ni ninguna extensión o te quedarás sin cuota de disco.

fjmadrid@uco.es

Trabajando con VSCode

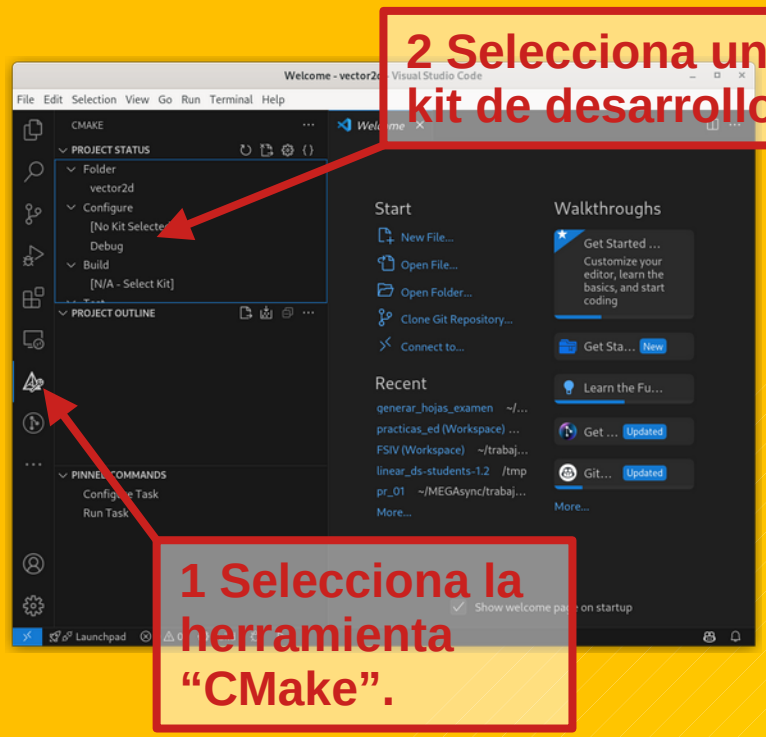
- Paso 1: abrir la carpeta con la práctica.

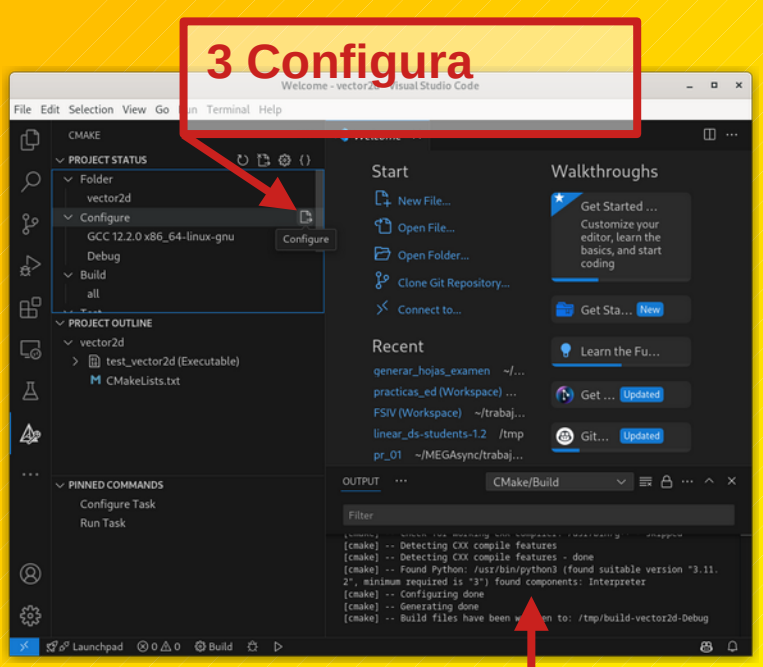


fjmadrid@uco.es

Trabajando con VSCode

- Paso 2: configura el proyecto.

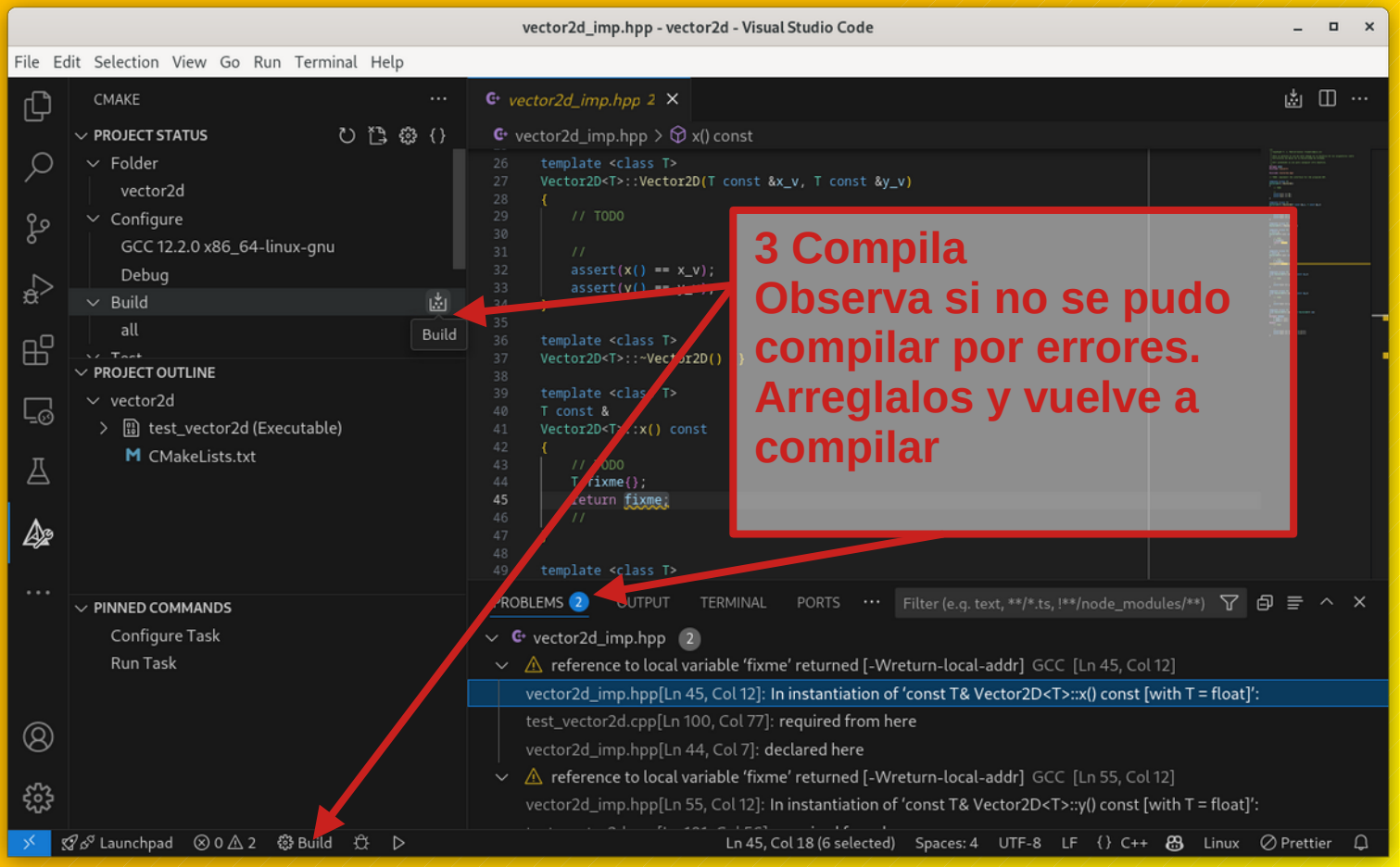




fjmadrid@uco.es

Trabajando con VSCode

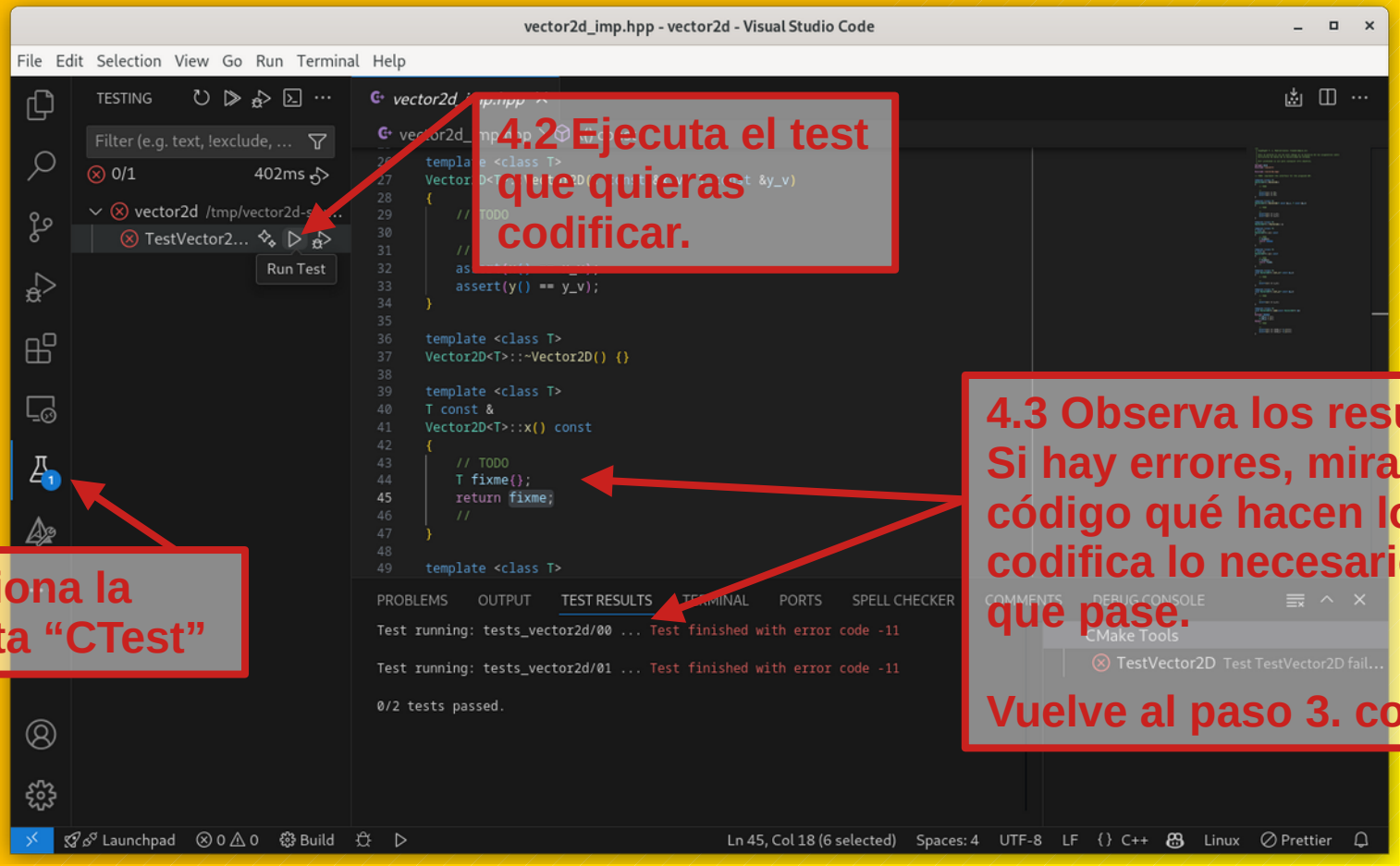
- Paso 3: compila el proyecto.



fjmadrid@uco.es

Trabajando con VSCode

- Paso 4: ejecuta los tests.



fjmadrid@uco.es

Trabajando con la línea de comandos

- Paso 1: configurar el proyecto.

```
# 0. Has descargado el zip con las fuentes en la carpeta “home”.

# 1. Crea una carpeta cuyo nombre sea tu login y entra en ella.
mkdir ma1macuf
cd ma1macuf

# 2. Descomprime el zip con las fuentes de las prácticas.
unzip ~/vector2d-students-1.1.zip

# 3. Entra en la carpeta con las fuentes.
cd vector2d

# 4a. Abre esta carpeta en VSCode y configurala usando el IDE.
# 4b. Configura en línea de comandos
cmake -S . -B build
```

fjmadrid@uco.es

Trabajando con la línea de comandos

- Paso 2: ciclo compilar/testear/codificar:

```
#5 Entra en el directorio de construcción.
cd build

#6 Compilar
make

#Si hay fallos de compilación, arregla los fallos y
#repite el paso 6.

#7 Ejecutar tests.
ctest -C Debug -T test --output-on-failure

#8a: Hay fallos: Codifica/Arregla fallos encontrados por los
     tests y vuelve al paso 6.

#8b: No hay fallos. !!Trabajo terminado!! ya puedes preparar
     la entrega. Sigue las instrucciones al respecto.
```

fjmadrid@uco.es

Repaso de conceptos C++

- Templates C++.
- Palabra clave “Typename”.
- Punteros inteligentes.
- Métodos estáticos de clase.
- Palabra clave “auto”.
- “Casting” en C++.

fjmadrid@uco.es

Repaso de conceptos C++

- Templates C++.

```
// vector2d.hpp

template < class T >
class Vector2D {
public :
  Vector2D(T x, T y);
  T x () const ;
  T y () const ;
private:
  T d_[2];
};

#include
<vector2d_imp.hpp>
```

```
// vector2d_imp.hpp

#include <vector2d.hpp>

template < class T >
T Vector2D<T>::Vector2D (T x,
                         T y)
{
  d_[0] = x;
  d_[0] = y;
};

template < class T >
T Vector2D<T>::x () const
{
  return d_[0];
};

template < class T >
T Vector2D<T>::y () const
{
  return d_[1];
};
```

```
// main.cpp

#include <vector2d.hpp>

int main(void)
{
  Vector2D<int> v1(0,0);
  Vector2D<float> v2(0.5,
-1.5);
  cout << “v1: (”
       << v1.x() <<’,’
       << v1.y() <<’)’
       << endl;
  cout << “v2: (”
       << v2.x() <<’,’
       << v2.y() <<’)’
       << endl;
}
```

fjmadrid@uco.es

Repaso de conceptos C++

- Palabra clave “typename”.
  - Necesaria para “ayudar” al compilador a distinguir entre un tipo de dato o un objeto cuando se escriben templates.

```
// vector2d.hpp

template < class T >
class Vector2D {
public :
  using Ref = Vector2D*;
  Ref clone ();
};
```

```
// vector2d_imp.hpp

template < class T >
Vector2D<T>::Ref clone () //ERROR
{
   return new Vector2D<T>(x(), y());
}
```

¿Vector2D<T>::Ref qué es?
...
no te preocupes será un tipo.

```
// vector2d_imp.hpp

template < class T >
typename Vector2D<T>::Ref clone ()
{
   return new Vector2D<T>(x(), y());
}
```

Repaso de conceptos C++

- Punteros inteligentes.
  - Gestionan la memoria dinámica.
  - Tipos: unique_ptr, weak_ptr, **shared_ptr**.

```
// main.cpp

int main()
{

    std::shared_ptr<Vector2D<int>> v1 = std::make_shared<Vector2D<int>>(1,1);

    // v1 se usa como si fuera un puntero a un objeto Vector2D.

    cout << "v1 : (" << v1->x() << ',' << v1->y() << ')' << endl;//v1: (1,1)
```

Repaso de conceptos C++

- Punteros inteligentes.
  - **shared_ptr** permite compartir un objeto.

```
// main.cpp

    std::shared_ptr<Vector2D<int>> v2 = v1;

    // Ahora v1 y v2 apuntan al mismo objeto Vector2D (lo comparten).

    cout << "v2 : (" << v2->x() << ',' << v2->y() << ')' << endl;//v2: (1,1)

    cout << "Asignado a v2.x el valor 3." << endl;

    v2->set_x(3);

    // Si cambio v2 cambia v1 también ya que comparten el mismo objeto.

    cout << "v1 : (" << v1->x() << ',' << v1->y() << ')' << endl;//v1: (3,1)
    cout << "v2 : (" << v2->x() << ',' << v2->y() << ')' << endl;//v2: (3,1)
```

fjmadrid@uco.es

Repaso de conceptos C++

- Punteros inteligentes.
  - El tipo `shared_ptr` libera la memoria cuando el contador de referencias llega a 0.
  - Consecuencia: no tenemos que preocuparnos de invocar “delete” para liberar la memoria reservada.

```
// main.cpp

    cout << "Liberamos v1" << endl;
    v1.reset(); //v2 sigue existiendo

    cout << "v2 : (" << v2->x() << ',' << v2->y() << ')' << endl;//v2: (3, 1)

    // Al terminar el ámbito de la variable v2, se destruye y como
    // ya no hay referencias compartidas al objeto “Vector2d” que apunta,
    // se libera también la memoria dinámica que se reservó
    // para almacenar dicho objeto Vector2D.

    return 0;
}
```

fjmadrid@uco.es

Repaso de conceptos C++

- Métodos estáticos de clase.
  - No necesitan un objeto para ser invocados.
  - Los usaremos para tener un constructor virtual.

```
// vector2d.hpp
template <typename T>
class Vector2D
{
...
public:
    static Ref create(T x, T y);
...
}
```

```
// vector2d_imp.hpp
template <typename T>
typename Vector2D<T>::Ref
Vector2D<T>::create(T x, T y)
{
    return new Vector2D<T>(x, y);
}
```

No es necesario tener un objeto Vector2D para usar create.

```
//main.cpp

    Vector2D<int>::Ref v7 = Vector2D<int>::create(1, 1);
    cout << "v7 : (" << v7->x() << ',' << v7->y() << ')' << endl;
```

Repaso de conceptos C++

- Palabra clave “auto”.
  - Es muy útil, úsala siempre que puedas.

```
//main.cpp

// En vez de:
   std::shared_ptr<Vector2D<int>> v5 = std::make_shared<Vector2D<int>>(-1, -1);

// podríamos haber escrito:
  auto v5 = std::make_shared<Vector2D<int>>(-1, -1);
```

Repaso de conceptos C++

- “Casting” en C++.
  - static_cast: coerciones en tiempo de compilación.

```
float v1 = 5.2;

// En vez de:
int v2 = (float) v1;

// Escribimos:

int v2 = static_cast<int>(v1);
```

Repaso de conceptos C++

- "Casting" en C++.
  - const_cast: permite quitar el calificador "const" a un objeto.

```
void UnTipo::un_metodo_const() const {

   auto this_ = const_cast<UnTipo*>(this);

   this->un_atributo_int = 0; //ERROR

   this_->un_atributo_int = 0;

}
```

`this` es "const" porque el método es "const" así que no podríamos modificar el atributo si no **quitamos** el calificador "const"

fjmadrid@uco.es

Estructuras de Datos
Grado de Ing. Informática

Introducción a las prácticas de EEDD

13/02/25