

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Métodos constantes

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid



# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Paso de objetos por valor

```
bool es_navidad(Fecha f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- La función `es_navidad` recibe su argumento **por valor**.
- Al pasar por valor una instancia de una clase se hace una copia del argumento.
  - ¿Cómo? Constructor de copia.
- Si queremos evitar eso, debemos pasar el parámetro **por referencia**.

# Paso de objetos por referencia



# Paso de objetos por referencia

```
bool es_navidad(Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- Mediante el símbolo & indicamos que el parámetro f se recibe por referencia.
- Con esto se evita hacer una copia de mi\_fecha.
- ¡Ojo! Cualquier cambio que es\_navidad realice en f se reflejará también en mi\_fecha.
  - En este caso, podemos ver que es\_navidad no está alterando el objeto f.

# ¿Y si no conocemos la implementación?

- ¿Cuál de estas dos funciones te inspira más confianza?

```
bool compara.Fecha f1, Fecha f2);
```

```
bool compara.Fecha &f1, Fecha &f2);
```

- La primera garantiza que no va a alterar el estado de los objetos Fecha que reciba, ya que va a trabajar sobre copias de los mismos.
- La segunda no ofrece esa garantía, aunque se ahorra la copia de los argumentos.
- **¿Podemos conseguir los beneficios de ambas versiones?**


# Referencias constantes

```
bool compara(const Fecha &f1, const Fecha &f2);
```

- Una **referencia constante** no permite modificar el estado del objeto apuntado por la referencia.
- El compilador comprueba que `compara` no modifique los atributos de los objetos `f1` y `f2`.
- Con esto:
  - Nos ahorramos copias de los argumentos, porque se pasan por referencia.
  - El que llame a la función `compara` tiene la certeza de que sus objetos no se van a ver modificados.

# Paso de objetos por valor

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}
```



- Hacemos que la función `es_navidad` reciba su parámetro como referencia constante.
- ... pero el compilador protesta sobre nuestra definición.
- El compilador no sabe si los métodos `get_dia()` o `get_mes()` alteran el estado de `f`.



# Métodos constantes



# Métodos constantes

```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
private:  
    ...  
};
```

- Se declaran añadiendo la palabra **const** tras la lista de parámetros.
- Con esto se indica que el método no altera el estado del objeto.
- El compilador comprueba:
  - que el método no modifique los atributos del objeto.
  - que el método no llame a otros métodos de ese mismo objeto, salvo que también sean constantes.


# Métodos constantes

```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};  
  
void Fecha::imprimir() const {  
    ...  
}
```

- Si un método se implementa fuera de la clase, es necesario poner `const` tanto en su declaración, como en su implementación.



# Llamadas a métodos constantes

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;   
}
```

- Si una referencia a un objeto es constante:
  - No podemos modificar sus atributos públicos a través de esa referencia.
  - Solamente podemos llamar a los métodos `const` de esa referencia.

# ¿Qué métodos deben ser const?

- Todos los que no modifiquen el estado del objeto que recibe la llamada al método (`this`).
- Este tipo de métodos reciben el nombre de **observadores**.
- Incluye, entre otros:
  - Métodos de acceso (`get`).
  - Métodos para imprimir el objeto por pantalla o a otro flujo de salida.
  - Métodos de conversión a otro objeto (por ejemplo, `to_string()`).