Deusto

PARTE 2

PROGRAMACIÓN EN C++

Asignatura

PROGRAMACIÓN IV

Deusto

TABLA DE CONTENIDOS

DECLARACIÓN Y USO DE CLASES

Deusto

EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

- Código **máquina**
- Lenguaje **ensamblador**
- **Tipos** e identificadores para datos
- **Funciones**
- **Agrupaciones** de datos
- Orientación a **objetos**
  - Encapsular datos y funciones
  - Abstraer la realidad de forma sencilla
  - Utilidades adicionales: sobrecarga, constructores y destructores, herencia, polimorfismo, etc.

Deusto

LENGUAJES C y C++

- Mucho de **lo visto en C es trasladable al lenguaje C++**
  - Estructuras de **control**
  - Declaración de **variables**
  - **Tipos** de variables básicas
  - **Estructuras** de datos
  - **Funciones**
  - **Punteros**
  - **Gestión de memoria** >> aunque en C++ se reserva y libera memoria de otra forma (no se usa *malloc* y *free*), pero la mecánica es similar.
  - **Módulos de código** >> separación en ficheros **.h** y **.cpp**

Deusto

LENGUAJE C++

- **Es otro lenguaje**
  - Se basa en la sintaxis del C pero **añade cosas nuevas**
- Diferencias
  - Compilador distinto >> **g++** en vez de gcc
  - Ficheros de código >> **.cpp** en vez de .c
  - **Orientación a objetos** >> herencia, polimorfismo, encapsulamiento…
- El proceso de generación de código es el mismo que en C
  - Pre-procesado
  - Compilación
  - Enlazado
  - Generación de ejecutable final

Deusto

CLASES (1)

- Una clase es una **estructura de datos y funciones**
  - **Datos miembro** >> a un dato miembro se le puede denominar **atributo**
  - **Funciones miembro** >> funciones que permiten ejecutar código dentro del ámbito de la clase y trabajar con los datos miembro así como otras funciones miembro. A una función miembro se le puede denominar **método**.
- Es posible controlar la **visibilidad** de los miembros (atributos y métodos)
  - Públicos
  - Privados
  - Protegidos

Deusto

CLASES (2)

- Ejemplo de declaración de una clase en C++

```
class Fecha
{
    // datos miembro de la clase
    unsigned int anyo;
    unsigned int mes;
    unsigned int día;

    bool esBisiesto()
    {
        return (this->anyo % 4 == 0)
            && (this->anyo % 100 != 0)
            || (this->anyo % 400 == 0);
    }
};
```

Deusto

EL PUNTERO THIS (1)

- Por cada **objeto instanciado**
  - **Diferente memoria** para los **datos** miembro
  - **Mismo código** para todas las **funciones** de los objetos
- ¿Cómo es capaz de saber qué datos miembro de qué objeto debe usar en el código de la función miembro?
  - C++ añade automáticamente un primer parámetro a cada función: un puntero constante al objeto >> **el puntero this**

```
class Fecha
{
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;

    bool esBisiesto([Fecha* const this])
    // así es como queda la función
    {
        return (this->anyo % 4 == 0)
            && (this->anyo % 100 != 0)
            || (this->anyo % 400 == 0);
    }
};
```

Deusto

EL PUNTERO THIS (2)

- El puntero **this** está **disponible para toda función miembro de una clase**
  - **Siempre** que dicha función **no sea estática**
- No es obligatorio usarlo siempre. Pero a veces es necesario:
  - **Diferenciar** entre el identificador de una variable local de la función miembro y el identificador de un dato miembro
  - **Referenciar** al objeto concreto por claridad

```
class Fecha
{
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;

    void setAnyo(unsigned int anyo)
    {
        this->anyo = anyo; //evitamos la confusión usando this
    }
};
```

Deusto

#INCLUDE <IOSTREAM> (1)

- La **escritura en consola** en C++ se hace utilizando el **operador <<**
  - **cout** es un objeto que representa la **salida a consola**
  - **endl** es el carácter de salto de línea
  - Es necesario incluir el **<iostream>**
  - Es posible incluir el **namespace std**

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

int main(void)
{
    int a = 5;
    cout << "El número es: " << a << endl; // escribimos
    return 0;
}
```

Deusto

#INCLUDE <IOSTREAM> (2)

- La **lectura de consola** se realiza utilizando el **operador >>**
  - **cin** es un objeto que representa la **lectura desde consola**
  - Es necesario incluir el **<iostream>**
  - Es posible incluir el **namespace std**

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

int main(void)
{
    int a;
    cout << "Escribe un número: ";
    cin >> a; // leemos desde consola a la variable a
    cout << endl;

    cout << "El número es: " << a << endl;
}
```

Deusto

ENCAPSULAMIENTO (1)

- Otra de las **utilidades de la POO** es el encapsulamiento
- Determina **si se permite o no al programador acceder a los miembros de la clase** (atributos o métodos) >> visibilidad
- Por ahora vamos a ver 2 niveles de **visibilidad**:
  - **Público:** los miembros son **siempre accesibles**
    - Palabra reservada >> ***public:***
  - **Privado:** los miembros **sólo son accesibles desde** el código de las funciones miembro de **la clase** y nunca desde cualquier otra función. Es la **visibilidad por defecto** si no se indica otra cosa.
    - Palabra reservada >> ***private:***

Deusto

ENCAPSULAMIENTO (2)

Fecha

```
class Fecha
{
// DATOS MIEMBRO
private:  // Si no se pone es lo mismo
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;

// FUNCIONES MIEMBRO
public:
    bool esBisiesto()
    {
        return (anyo % 4 == 0) && (anyo % 100 != 0) || (anyo % 400 == 0);
    }
};
```

Main

```
void main()
{
    Fecha f;
    f.anyo = 2006;  // Error de compilación. El atributo no es público
    f.mes = 11;       // Error de compilación. El atributo no es público
    f.dia = 12;           // Error de compilación. El atributo no es público
    cout << f.esBisiesto() << endl;   // Compilación correcta
}
```

Deusto

ENCAPSULAMIENTO (3)

- Es muy recomendable que los **atributos nunca sean públicos**
  - Acceso mediante métodos >> eso es **encapsular**
- Para cambiar/obtener sus valores
  - Métodos "**setXXX**" y "**getXXX**"

Deusto

CONSTRUCTORES

- ¿Cómo se puede **asignar valores iniciales a los atributos** cuando se instancia un objeto?
  - Existe una función miembro especial que se ejecuta cuando se instancia un objeto llamada **constructor**
  - Es un **método sin valor de retorno y llamado como la clase**

Fecha

```
class Fecha
{
private:
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;
public:
    Fecha()
    {
        anyo = 2006;
        mes = 1;
        dia = 1;
    }
};
```

Main

```
void main()
{
    /* Se llama al constructor y se inicializan
    los valores a 1/1/2006 */
    Fecha f;
}
```

Deusto

DESTRUCTORES (1)

- Si hemos reservado memoria/recursos en el constructor de un objeto…
  - ¿Cómo la liberamos cuando ya no hace falta?
- En C++ existe el concepto de **destructor** de una clase
- El destructor es una función miembro que:
  - **No recibe argumentos**
  - **No devuelve nada**
  - Se llama exactamente igual que la clase con la virgulilla **'~' por delante**
- El uso más común para los destructores es el de **comprobar que los recursos que el objeto tenía asignados han sido liberados**
  - Liberar memoria, cerrar un fichero, etc.

Deusto

DESTRUCTORES (2)

- Los destructores en C++ **son llamados al eliminar el objeto**
- **Objetos creados automáticamente >> cuando termina el ámbito** de uso de dicho objeto (al salir de la función donde se ha declarado del objeto)

```
class Alumno;

void main(void)
{
    Alumno a; //aquí se llama al constructor sin parámetros
} // al finalizar la función se llama al destructor
```

- **Objetos creados dinámicamente >> cuando se llama explícitamente a la función para liberar memoria** en C++
  - En C++ las instancias se liberan con **delete**

Deusto

DESTRUCTORES (3)

Fecha

```
class Fecha
{
private:
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;
public:
    Fecha()
    {
        anyo = 2006;
        mes = 1;
        Dia = 1;
        [. . .]  // operaciones que reservan recursos: memoria, ficheros, etc.
    }
    ~Fecha()
    {
        [. . .]  // operaciones a realizar cuando se destruye el objeto + liberar recursos
    }
};
```

Main

```
void main()
{
    Fecha f; // Se llama al constructor y se inicializan los valores a 1/1/2006
} // Se llama al destructor al eliminar el objeto f
```

Deusto

CONSTRUCTORES Y DESTRUCTORES

- Para cada clase existe **un constructor (sin argumentos) y destructor "por defecto"**
  - Realizan una **función** de asignación de memoria, inicialización (constructor) y liberación (destructor) **básica**
  - **Están presentes mientras el programador no aporte un constructor y destructor propios**
- Tanto el constructor como el destructor son funciones miembro por lo que se ven afectadas por el encapsulamiento
- ¿Qué pasa si declaramos un constructor y/o destructor privado?
  - Sólo se podrían instanciar objetos de la clase desde dentro de código de la propia clase
  - Sólo se podrían destruir las instancias de la clase desde dentro del código de la propia clase (por ejemplo, métodos estáticos)
- En principio, **todos los constructores y destructores serán públicos**

Deusto

OPERADOR DE RESOLUCIÓN DE ÁMBITO

- Podemos **implementar el código de las funciones miembro fuera de la declaración de la clase**
  - Se utiliza el **operador ::** para indicar a qué clase pertenece una función implementada fuera
  - Separar declaración e implementación
    - **Fichero de cabecera o .h** >> se suele declarar la clase
    - **Fichero fuente o .cpp** >> implementa las funciones miembro haciendo uso del operador de ámbito
- Permite más legibilidad además de ocultar el código

fecha.h

```
class Fecha
{
    unsigned int anyo;
    unsigned int mes;
    unsigned int dia;
public:
    Fecha();
    void setAnyo(unsigned int anyo);
};
```

fecha.cpp

```
Fecha::Fecha()
{
    anyo = 2006;
    mes = 1;
    dia = 1;
}
void Fecha::setAnyo(unsigned int anyo)
{
    this->anyo = anyo;
}
```

Deusto

MIEMBROS ESTÁTICOS (1)

- Los **miembros (datos y funciones)** pueden ser **estáticos**
  - Palabra reservada delante de la declaración >> **static**
- Indica que **ese miembro es único para todas las instancias** (objetos) que se creen a partir de esa clase
  - **Atributos estáticos** >> deben declararse y, opcionalmente, asignarles un valor inicial
  - **Métodos estáticos** >> desde una función miembro estática sólo se tiene acceso a miembros estáticos
- **Para poder hacer uso de los miembros estáticos no se necesita una instancia u objeto**
  - Se invocan usando el **operador de resolución de ámbito**

Deusto

MIEMBROS ESTÁTICOS (2)

- **Declaración y asignación** de variables estáticas

```
class Fecha
{
public:
    static const unsigned int MESES;
    static bool esBisiesto(unsigned int anyo);
};
const unsigned int Fecha::MESES = 12;
bool Fecha::esBisiesto(unsigned int anyo)
{
    return (anyo % 4 == 0) && (anyo % 100 != 0) || (anyo % 400 == 0);
}
```

- **Acceso** a miembros estáticos

```
void main()
{
    cout << "Fecha::MESES = " << Fecha::MESES << endl;
    cout << "¿2006 es bisiesto? " << (Fecha::esBisiesto(2006)?"Si":"No") << endl;
}
```

Deusto

NAMESPACES (1)

- C++ proporciona mecanismos para **organizar y agrupar** las clases, entidades y funciones
  - Se puede definir **namespaces**
  - Permiten **evitar colisiones de nombres** entre distintas partes de un proyecto

```
namespace geom // Nombre del namespace >> geom
{
  [. . .] // Declaraciones de clases, variables, funciones
    class Point
    {
        int x, y;
        [. . .]
    };
}
```

- Pueden existir **clases que se llamen igual** pero **deben estar en espacios de nombres distintos**

Deusto

NAMESPACES (2)

- Cuando queremos utilizar nombres contenidos en un espacio de nombres tenemos que **indicar el namespace al que pertenecen**

```
geom::Point * p = new geom::Point[5]; // Usamos el Point de geom y no otro
```

- Para evitar repetir el nombre del namespace cada vez que usamos una clase, variable o función definida >> **using namespace name**;
  - Sería equivalente al import de Java
- Las utilidades de **<iostream>** están incluidas en el **namespace std**
  - Podemos evitar tener que hacer referencia constantemente a std, diciendo que usamos el namespace std

```
#include <iostream>
using namespace std; //evita tener que poner std:: al usar el cout de iostream

int main(void)
{
    cout << "Hola, mundo" << endl; // Frente a poner std::cout
}
```

Deusto

GESTIÓN DE MEMORIA (1)

- El operador **new** permite **reservar memoria (dinámica)** para distintos elementos (tipos primitivos, estructuras y clases)

```
TIPO *p = new TIPO();
int *p = new int(); //reservamos memoria para un entero
```

- Se pueden definir **constructores con parámetros**

```
class Point
{
   int x, y;
public:
   Point(int x, int y)
   {
     this->x = x;
     this->y = y;
   }
};

Point *p = new Point(2, 3); //memoria para un objeto Point
```

Deusto

GESTIÓN DE MEMORIA (2)

- El **programador** es responsable de **liberar la memoria** que ha **reservado con new**
- Se utiliza el operador **delete**

```
delete p; // p es un puntero al objeto reservado con new
```

- El programador debe **mantener la dirección la memoria reservada** para poder liberarla correctamente
- **Una vez que la memoria de un objeto ha sido liberada NO se debe usar dicho objeto**
  - Si lo usamos por error a través de su puntero el programa puede parecer que funciona (mientras que en Java tendríamos un NullPointerException)
  - Tendremos **errores difíciles de encontrar**

Deusto

GESTIÓN DE MEMORIA (3)

- **Reserva de memoria para arrays**
  - Es posible reservar bloques de memoria consecutivos con el operador **new**

```
tipo *p = new tipo[N]; //reservamos N variables
```

  - Ejemplo:

```
class Point
{
    int x, y;
public:
    Point() { this->x = 0; this->y = 0};
};
Point *p = new Point[5]; //reservamos 5 objetos Point
```

- Debemos indicar a **delete** que se trata de un array **usando []**

```
delete[] p;    // liberamos todo el array, NO solamente un elemento: delete p;
```

Deusto

EJERCICIOS

Programación en C++: Enunciados

- Ej01: Clases
- Ej02: Clases

REFERENCIAS

Deusto

REFERENCIAS (1)

- Es un **tipo de datos especial** que representa referencias a un objeto o tipo primitivo
- Se declaran con **&** (se lee como referencia a tipo)
  - **NO confundir con el operador & que obtiene la dirección de una variable**
- **No son punteros** >> puede parecer algo similar pero NO es lo mismo
  - Java trabaja con referencias (no tiene punteros), C++ tiene ambas cosas
- Una referencia es como **un segundo nombre para una variable**
- Las referencias **siempre deben ser inicializadas**
  - No pueden ser NULL

```
int a = 5;

int &b = a; //tanto b como a son la misma variable con dos nombres
b = 5;      // a y b ahora valen 5
```

Deusto

REFERENCIAS (2)

- Permiten **evitar el paso por valor** en funciones de una **forma más sencilla que utilizando punteros**

```
void sumar(int *a, int *b)
{
    *a = *a + *b;
}
int a = 5;
int b = 7;
sumar(&a, &b); //al pasar punteros/direcciones podemos modificar
```

- Podemos resolver **este mismo problema con referencias**

```
void sumar(int &a, int &b) //recibimos la variable como ref.
{
    a = a + b; //modificamos la variable exterior. NO COPIA.
}
int a = 5;
int b = 7;
sumar(a, b); // se pasa la referencia a las variables enteras
```

Deusto

REFERENCIAS (3)

- Aparte de lo expuesto anteriormente, la principal razón para usar referencias podemos observarla en el siguiente ejemplo:

```
class Cadena
{
private:
    char* cadena;
public:
    Cadena()
    {
        cadena = new char[1];
        cadena[0] = '\0';
    }
    Cadena(char* c)
    {
        cadena = new char[strlen(c) + 1];
        strcpy(cadena, c);
    }
    ~Cadena()
    {
        delete [] cadena;
    }
    bool igualA(Cadena c)
    {
        return strcmp(cadena, c.cadena) == 0;
    }
    void presenta()
    {
        cout << cadena << endl;
    }
};
```

Deusto

REFERENCIAS (4)

- La función miembro que nos interesa es **"igualA"**:

```
void main()
{
    Cadena c1("Una cadena");
    Cadena c2("Otra cadena");
    c1.presenta();
    c2.presenta();
    cout << (c1.igualA(c2)?"Igual":"Distinta") << endl;
    c1.presenta();
    c2.presenta();
};
```

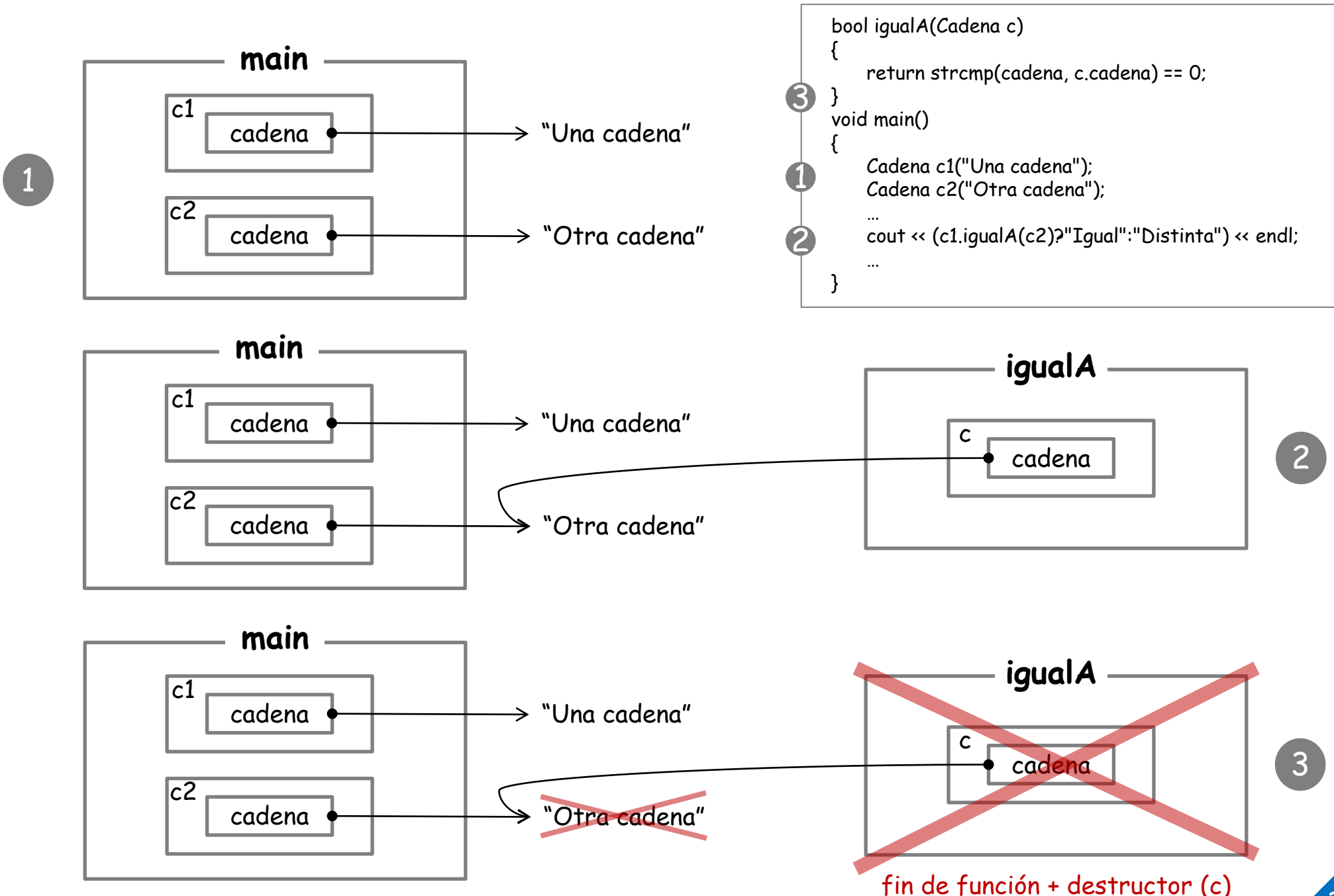
¿Funcionaría bien este programa (clase + main)?

Deusto

REFERENCIAS (5)

- Si se ejecuta el programa expuesto, se conseguirá un error en tiempo de ejecución dado que la llamada a la función **"presenta"** del objeto **"c2"** después llamar al método **"igualA"** estará accediendo a memoria liberada (aunque el **puntero "cadena"** no lo sabe).
- El problema consiste en:
  - Cuando se hace la llamada desde el **main** a la función miembro **"igualA"** se realiza una **copia en memoria del objeto "c2"** del main **al objeto "c"** de la función miembro. Esta copia por defecto lo que hace es **copiar byte a byte los datos miembro de ambos objetos**. Sólo se copian los punteros (sus direcciones de memoria). **No se reserva nueva memoria** (no se llama al constructor definido por nosotros), sino que se comparte.
  - Cuando el **ámbito de la función miembro termina**, el **objeto "c"** se debe eliminar y se llama al **destructor** por lo que la memoria se libera con el **delete**. Pero el objeto del **main** no se ha enterado y **su puntero sigue creyendo que apunta correctamente a memoria que ha sido liberada**.

REFERENCIAS (6)



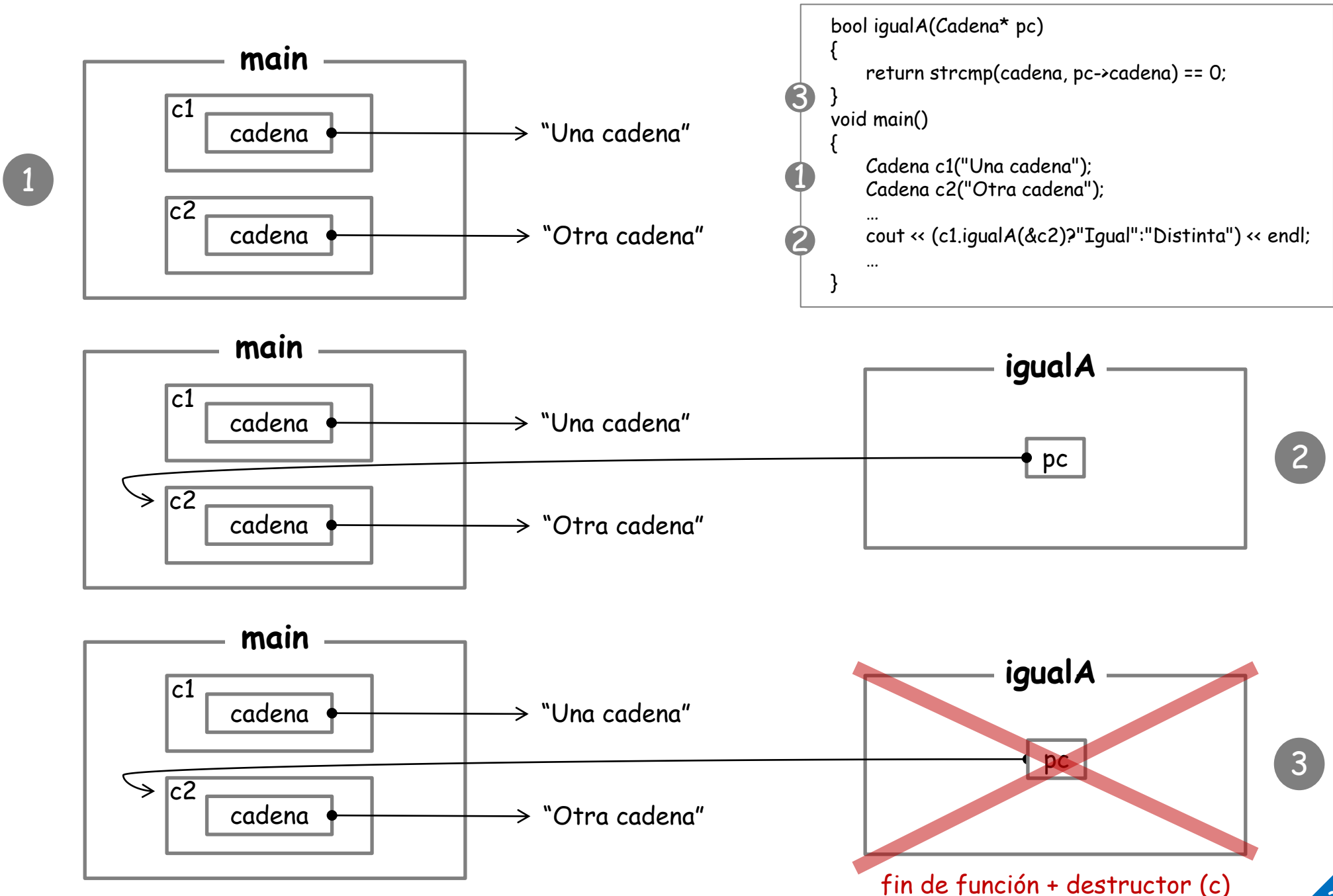
Deusto

REFERENCIAS (7)

- **Con punteros** solucionaríamos el problema ya que no trabajaríamos con copias de objetos sino simplemente copias de punteros.

```
...
bool igualA(Cadena* pc)
{
    return strcmp(cadena, pc->cadena) == 0;
}
...
void main()
{
    Cadena c1("Una cadena");
    Cadena c2("Otra cadena");
    c1.presenta();
    c2.presenta();
    cout << (c1.igualA(&c2)?"Igual":"Distinta") << endl;
    c1.presenta();
    c2.presenta();
}
```

REFERENCIAS (8)



Deusto

REFERENCIAS (9)

- Sin embargo, **con referencias** el problema queda mucho más elegante con los mismos resultados que con punteros

```
...
bool igualA(Cadena& c)
{
    return strcmp(cadena, c.cadena) == 0;
}
...
void main()
{
    Cadena c1("Una cadena");
    Cadena c2("Otra cadena");
    c1.presenta();
    c2.presenta();
    cout << (c1.igualA(c2)?"Igual":"Distinta") << endl;
    c1.presenta();
    c2.presenta();
}
```

Deusto

ÁMBITO DE VARIABLES Y REFERENCIAS

- Es posible que una **función retorne una referencia**
  - Hay que tener **cuidado de que la variable devuelta siga existiendo**

```
int& function(int a, int b);
{
    int c;
    return c; //referencia a algo que solo existe aquí
}  // la variable dejará de existir en este punto >> ERROR
```

  - Si el **ámbito de la variable es externo a la función**

```
int& function(int &a, int b);
{
    return a; //la referencia viene de fuera
}  //la variable no dejará de existir al terminar la función >> OK
```

EJERCICIOS

Programación en C++: Enunciados

- Ej03: Referencias

CONST

Deusto

MODIFICADOR CONST

- C++ permite al desarrollador **indicar que algo va a ser constante por diseño del programa**
  - El compilador comprueba que algo NO debe poder ser modificado
  - Facilita encontrar errores lógicos durante la compilación
- En C++ podemos usar la palabra reservada **const** con
  - **Variables** >> indica que el contenido de la variable no podrá ser modificado
  - **Métodos** >> indica que el método no modifica el estado de la clase a la que pertenece

Deusto

VARIABLES CONST

- Si lo aplicamos a **tipos simples**, indica que la variable tiene un **valor asignado que no se modificará** en todo lo que dure su ámbito

```
const int a = 5;  // a no podrá cambiar nunca de valor
a = 3;            //error en compilación
```

- Punteros >> Leer de derecha a izquierda para entender la sintaxis
  - **Punteros constantes a un objeto** >> no se puede modificar el puntero.

```
int* const p; // p es un puntero constante a un entero NO constante
```

  - **Punteros a un objeto constante** >> no se puede modificar el objeto

```
const int* p; // p es un puntero NO constante a un entero constante
```

  - **Ambos a la vez** >> no podemos modificar el puntero ni el objeto

```
const int* const p; // p es un puntero constante a un entero constante
```

Deusto

REFERENCIAS CONST

- **Las referencias son constantes por defecto**

```
int & const a // no tiene sentido. La variable a ya es una ref. constante
```

- Pero sí **podemos indicar si lo referenciado puede ser modificado o no**
- **Referencia a objeto constante** >> no podemos modificar el objeto
  - Evitamos que se copie/modifique al pasarlo como argumento

```
const int & a // a es una ref. a un entero constante
```

- **Es muy común usar referencias a objetos constantes** en el código de un programa en C++
  - Evitamos usar punteros
  - Tenemos la ventaja de evitar copias de los parámetros
  - Aseguramos que por error no modificamos los argumentos

Deusto

MÉTODOS CONST (1)

- Al declarar un **método** de una clase como **const** indicamos que **NO va a modificar el objeto this correspondiente**
  - Sobre un objeto const solo podemos llamar métodos const

Deusto

MÉTODOS CONST (2)

Point

```
class Point
{
    int x, y;
public:
    int getX() const // Garantizamos que no modifica el objeto
    {
        return x;
    }
    void setX(int x) // No ponemos const porque queremos modificar el objeto
    {
        this->x = x;
    }
};
```

```
void funcion(Point& p1, const Point& p2)
{
    int a = p1.getX();   // Correcto. Llamamos a un método sobre objeto no-const
    int b = p2.getX();   // Correcto. Llamamos a un método const sobre objeto const
    p1.setX(10);  // Correcto. Llamamos un método no-const sobre un objeto no const
    p2.setX(10);  // Error. Estamos llamando a un método no-const (que modifica un objeto const)
}
```

Deusto

EJERCICIOS

Programación en C++: Enunciados

- Ej04: Const

CONSTRUCTOR COPIA

Deusto

ASIGNACIÓN Y COPIA DE OBJETOS (1)

- Supongamos que tenemos la siguiente clase:

```
class Alumno
{
    int id;
    int numNotas;
    int * notas;

public:
    Alumno(const int id, const int numNotas)
    {
        this->id = id;
        this->numNotas = numNotas;
        this->notas = new int[numNotas];
    };

    void setNota(int index, int nota)
    {
        notas[index] = nota;
    }
    int getNota(int index)
    {
        return notas[index]
    };

    ~Alumno()
    {
        delete[] notas;
    }
};
```

Deusto

ASIGNACIÓN Y COPIA DE OBJETOS (2)

- ¿Qué ocurre si hacemos lo siguiente?

```
Alumno a(1,5);
[. . .] // operamos con 'a' modificando notas
Alumno b = a;
[. . .] // operamos con b modificando sus notas
imprimir(a);
imprimir(b);
```

- ¿Es correcto el código anterior? NO >> Fallo bastante complicado de encontrar en ejecución.
  - Los dos objetos **comparten datos** (notas) **por error**
  - Lo que es peor >> si una de las variables Alumno es liberada también se liberaran los datos de la otra >> ***memory leak***

Deusto

ASIGNACIÓN Y COPIA DE OBJETOS (3)

- ¿Por qué ha ocurrido esto?
  - La **copia y asignación en C++** de objetos es **byte a byte**
    - C++ copia el contenido de las variables una a una
  - Para **tipos simples** no hay problema **>> copia su contenido**
    - Por ejemplo, el id de Alumno
  - En el caso de **punteros** copiara también **su contenido (dirección apuntada por el puntero) pero NO lo apuntado por dicho puntero**
- ¿Cómo podemos solucionar esto?
  - Primero **tenemos que entender qué ocurre en C++** cuando:
    - Se **construye** un objeto **por copia**
    - Se **asigna** un objeto a otro

Deusto

CONSTRUCTOR COPIA (1)

- Los objetos tienen un **constructor que se llama cuando un objeto se construye como copia de otro >> constructor copia**
- Los **casos donde se llama** son los siguientes:
  - Cuando se declara un objeto y se hace una **llamada explícita** a ese constructor

```
Alumno a(1,5);
Alumno b(a); // Esto es una llamada explícita al constructor copia
```

  - Cuando se declara un objeto y se hace una **asignación en ese mismo instante**

```
Alumno a(1,5);
Alumno b = a; // Se llama al constructor copia
```

  - Cuando se pasa un **objeto a una función por valor, o se retorna**

```
Alumno funcion(Alumno a) // copia al pasar argumento valor
{
    Alumno b(2,4);
    return b; // constructor copia al retornar el objeto
}
```

Deusto

CONSTRUCTOR COPIA (2)

- **Por defecto** el constructor copia hace una **copia byte a byte**
- Tenemos que **implementarlo nosotros** si queremos que se hagan **copias profundas** de los objetos (deep copy).
- El constructor copia se define de la siguiente forma:

```
NombreClase(const NombreClase &o); //ref. a objeto constante
```

- Por ejemplo, para la clase Alumno el constructor copia podría ser:

```
Alumno(const Alumno &a)
{
   // Realizamos una copia de las estructuras internas en memoria
   this->id = a.id;
   this->numNotas = a.numNotas;
   this->notas = new int[numNotas];
   for (int i = 0; i < numNotas; i++)
   {
      this->notas[i] = a.notas[i];
   }
}
```

Deusto

ASIGNACIÓN

- Cuando se realiza una **asignación** utilizando el **operador =**

```
Alumno a(1,5);
Alumno b(2,4);

a = b; //esto llama al operador de asignación
```

- **Por defecto**, la asignación de objetos también se hace **byte a byte**
- El **programador debe dar un comportamiento a sus clases** para la asignación **si queremos que ocurra algo diferente**
  - C++ no sabe hacer esas cosas si no le decimos qué asignar
- Veremos cómo se resuelve este problema más adelante
  - Utilizando la **sobrecarga de operadores** en C++

EJERCICIOS

Programación en C++: Enunciados

- Ej05: Constructor copia

HERENCIA

Deusto

HERENCIA (1)

- La herencia permite establecer **relaciones de especificación / generalización** entre dos o más clases
- Conceptualmente, una **clase hija** extiende o **modifica la funcionalidad de la clase padre**
- Cuando se **instancia** una clase que pertenece a una jerarquía de herencia
  - Se crea **memoria** para los atributos del **objeto** y para aquellos **heredados**
  - Se crean las estructuras de memoria para invocar métodos
    - De la clase instanciada
    - Correspondientes a las clases padre
- En C++ existe **herencia múltiple** >> una clase puede heredar de varias
  - Mediante un correcto diseño se puede evitar esta necesidad
- En C++ se indica que una clase hereda de otra/s así:

```
class Derivada: public ClaseBase1, public ClaseBase2, . . .
```

Deusto

HERENCIA (2)

Base

```
#include <iostream>
using namespace std;
class Base
{
private:
    float f1;
public:
    Base()
    {
        cout << "Base::Base()" << endl;
    }
};
```

Derivada

```
class Derivada: public Base
{
private:
    float f2;
public:
    Derivada()
    {
        cout << "Derivada::Derivada()" << endl;
    }
};
```

Main

```
int main()
{
    cout << "sizeof(Base) = " << sizeof(Base) << endl;
    cout << "sizeof(Derivada) = " << sizeof(Derivada) << endl;
    Derivada derivada;
    return 0;
}
```

Deusto

ENLACE DINÁMICO (1)

- Debido a que internamente se crean **varios objetos al instanciar** una clase derivada
  - Podemos **referirnos a cualquiera de ellos**
  - Pero **necesitamos utilizar punteros** para poder hacer eso

```
Derivada derivada; // Se crean dos objetos seguidos en memoria
Derivada* pDerivada = &derivada;
Base* pBase = &derivada;
```

- Esto se llama **enlace dinámico**
  - Si tenemos un **puntero a una clase Base**
    - Podremos **referenciar** a instancias de dicha clase
    - O **a instancias de clases hijas**

Deusto

ENLACE DINÁMICO (2)

- Podríamos crear un **array de punteros al tipo general** y guardar referencias a objetos de distintas clases derivadas

Jerarquía

```
class Base
{
   [. . .]
}
class Derivada1 : public Base
{
   [. . .]
}
class Derivada2 : public Base
{
   [. . .]
}
```

Main

```
int main()
{
   Base* pBase[3];
   pBase[0] = new Base();
   pBase[1] = new Derivada1();
   pBase[2] = new Derivada2();
}
```

Deusto

MODIFICADOR PROTECTED

- Una sección de una clase (atributos o métodos) puede ser
  - **private** >> solamente accesible desde instancias de la clase
  - **public** >> accesible desde fuera de la clase
  - **protected** >> accesible por instancia de la clase y por instancias de clases derivadas

Base

```
class Base
{
private: // Solo accesibles desde esta clase
    int a;
    void methodA() { … }
protected: // Accesibles desde esta clase e hijas
    int b;
    void methodB() { … }
};
```

Derivada

```
class Derivada: public Base
{
public:
    Derivada()
    {
        this->b = 3;
        this->methodB();
    }
};
```

Deusto

MODIFICADORES DE HERENCIA

- Al heredar podemos indicar la visibilidad de la herencia
  - Si heredamos con **public**

    > class Derivada: public Base

    - Todos los métodos mantienen su visibilidad
  - Si heredamos con **protected**

    > class Derivada: protected Base

    - Las secciones *public*: se heredan como *protected*:
    - Secciones *private*: se quedan igual
  - Si heredamos con **private**

    > class Derivada: private Base

    - Las secciones *public*: se heredan como *private*:
    - Las secciones *protected*: se heredan como *private*:
    - Las secciones *private*: quedan igual

Deusto

CONSTRUCTORES Y HERENCIA

- ¿Cómo indicamos **qué constructores de la clase padre deben llamarse**?
  - Especialmente si hemos eliminado el constructor por defecto de la clase base
- Utilizamos la **lista de inicialización al final del constructor de la clase derivada** para indicar que constructor de la clase padre llamar

```
CONSTRUCTOR(...): LISTA_DE_INICIALIZACION { ... }
```

- También pueden usarse para inicializar variables
  - Nos **evitamos** los **this->variable** en el constructor y los ponemos en una lista
  - Son más eficientes porque no crean la variable vacía y luego le asignan el valor
    - Se produce la creación con el valor directamente

Deusto

LISTA DE INICIALIZACIÓN

Base

```
class Base
{
private:
    float f1;
public:
    Base(float f1): f1(f1) // Lista de inicialización. this->f1 = f1;
    {
        cout << "Base::Base(float)" << endl;
    }
};
```

Main

```
int main()
{
    Derivada derivada(5.5f);
    return 0;
}
```

Derivada

```
class Derivada: public Base
{
private:
    float f2;
public:
    Derivada(float f1): Base(f1), f2(0) // Lista de inicialización. Llamada al constructor Base y this->f2 = 0
    {
        cout << "Derivada::Derivada(float)" << endl;
    }
};
```

Deusto

EJERCICIOS

Programación en C++: Enunciados

- Ej06: Herencia

POLIMORFISMO

Deusto

POLIMORFISMO (1)

- Permite **invocar un método sobre objetos de distinto tipo** relacionados mediante herencia y que se elija dinámicamente el método correcto
- **Por defecto los métodos en C++ NO son polimórficos**
- Se debe indicar mediante el **modificador "virtual"** delante del método para convertirlos en polimórficos

```
virtual METODO(...) { }
```

- El polimorfismo no es más que un **cambio de puntero a función en tiempo de ejecución**
- Necesitamos que los **objetos estén referenciados mediante punteros para poder usar el polimorfismo**
  - En Java los objetos (referencias) son punteros (escondidos) y por eso funciona el polimorfismo por defecto para todos los objetos

POLIMORFISMO (2)

Base

```
class Base
{
private:
    float f1;
public:
    void metodoNoPolimorfico()
    {
        cout << "void Base::metodoNoPolimorfico()" << endl;
    }
    virtual void metodoPolimorfico()
    {
        cout << "void Base::metodoPolimorfico()" << endl;
    }
};
```

Derivada

```
class Derivada: public Base
{
private:
    float f2;
public:
    void metodoNoPolimorfico()
    {
        cout << "void Derivada::metodoNoPolimorfico()" << endl;
    }
    virtual void metodoPolimorfico()
    {
        cout << "void Derivada::metodoPolimorfico()" << endl;
    }
};
```

Main

```
int main()
{
    Base* pBase = new Derivada();
    pBase->metodoNoPolimorfico(); // al ser no poliformico se llama a la implementación del tipo del puntero (Base)
    pBase->metodoPolimorfico(); // como el método es polimórfico se llama a la implementación del tipo del objeto (Derivada)
    delete pBase;
    return 0;
}
```

Deusto

POLIMORFISMO (3)

- Para **llamar a una implementación concreta de un método polimórfico,** desde un método derivado a un método de la clase base se debe hacer uso del **operador de resolución de ámbito**

```
Base::metodoPolimorfico();
```

- Podemos **reutilizar código** de métodos polimórficos desde las clases derivadas

Base

```
class Base
{
private:
    float f1;
public:
    […]
    virtual void metodoPolimorfico()
    {
        cout << "void Base::metodoPolimorfico()" << endl;
    }
};
```

Derivada

```
class Derivada: public Base
{
private:
    float f2;
public:
    […]
    virtual void metodoPolimorfico()
    {
        // llamada a implementación del método en la clase Base
        Base::metodoPolimorfico();
        cout << "void Derivada::metodoPolimorfico()" << endl;
    }
};
```

Deusto

DESTRUCTORES Y HERENCIA

- Cuando un objeto se destruye
  1. Se llama a **su destructor**
  2. Se llama **al destructor de la clase padre**
  3. Se **va ascendiendo por la jerarquía de clases** llamando a los destructores
- Hay que tener en cuenta que los **destructores no son polimórficos por defecto**
  - Si usamos punteros a objetos y mezclamos tipos deberíamos indicar que los **destructores** son ***virtual***
  - Así, al destruir el objeto se **llamará al destructor correcto y no al de la clase base**
    - En caso contrario >> partes del objeto sin destruir
  - Por norma general >> **hacer que los destructores sean virtuales**

Deusto

DESTRUCTORES POLIMÓRFICOS

Base

```
class Base
{
private:
    float f1;
public:
    virtual ~Base()
    {
        cout << "Base::~Base()" << endl;
    }
};
```

Derivada

```
class Base
{
private:
    float f1;
public:
    virtual ~Base()
    {
        cout << "Base::~Base()" << endl;
    }
};
```

Main

```
int main()
{
    Base* pBase = new Derivada();
    delete pBase; // Llama al destructor correcto (Derivada)
    return 0;
}
```

EJERCICIOS

Programación en C++: Enunciados

- Ej07: Polimorfismo

CLASES ABSTRACTAS

Deusto

MÉTODOS ABSTRACTOS

- Para indicar que una **clase es abstracta** (no puede instanciarse) debemos indicar que **uno o más de sus métodos son abstractos**
- En C++ para declarar un método abstracto debemos indicar que es un **método virtual puro**
  - Es decir, virtual que no tiene ninguna implementación (= 0)

```
virtual void metodo() = 0;
```

- Para que las clases derivadas puedan instanciarse deben implementar todos los métodos heredados que son abstractos

Deusto

MÉTODO VIRTUAL PURO

Base

```
class Base
{
private:
   float f1;
public:
   virtual void metodo() = 0;
};
```

Derivada

```
class Derivada: public Base
{
private:
   float f2;
public:
   virtual void metodo()
   {
      cout << "void Derivada::metodo()" << endl;
   }
};
```

Main

```
int main()
{
   Base* pBase = new Derivada();
   pBase->metodo() // se llama a la implementación de la clase Derivada
   delete pBase;
   return 0;
}
```

Deusto

EJERCICIOS

Programación en C++: Enunciados

- Ej08: Clases abstractas

SOBRECARGA DE OPERADORES

Deusto

SOBRECARGA DE OPERADORES (1)

- **Sobrecarga** de métodos/funciones en un lenguaje de programación
  - Posibilidad de **tener métodos/funciones con el mismo nombre**
    - Siempre que tengan **distinto número/tipo de parámetros**
    - El tipo del **retorno no se tiene en cuenta**
- **En C++** es posible aplicar el concepto de la **sobrecarga a operadores**
  - Por ejemplo, el operador suma +

```
Point *p1 = new Point(2, 3);
Point *p2 = new Point(4, 7);

Point p3 = *p1 + *p2 //Sobrecargado el operador + para puntos. p3 vale (6, 10)
```

Deusto

SOBRECARGA DE OPERADORES (2)

- Podemos sobrecargar **casi cualquier operador** del lenguaje C++
  - Aritméticos → ++a, a++, a + b, a – b, a / b, a % b, etc.
  - Relacionales → a < b, a <= b, a > b, a == b, a != b, etc.
  - Operadores de bit → a << b, ~a, a & b, etc.
  - Otros operadores:
    - Asignación básica: a = b
    - Llamada a función: a()
    - Índice de array: a[b]
    - Operador *new*
    - Operador *delete*

Deusto

SOBRECARGA DE OPERADORES (3)

- Para **sobrecargar un operador** que pueda ser aplicado sobre objetos de una clase
  - Es necesario **implementar un método/función especial** para que C++ sepa como aplicar un operador a un tipo concreto
  - Podemos hacerlo de **dos formas**:
    1. Como **función miembro (método)**
    2. Como **función no miembro (friend)**
  - En cualquiera de los dos casos es necesario sobrecargar una función de la forma:

```
tipo operatorXX(tipo1, ...)
```

Deusto

SOBRECARGA COMO MÉTODO (1)

- Si un operador se **sobrecarga como método de una clase**
  - Si se trata de un **operador binario**: a OP b
    - El **primer argumento** es el puntero **this** (no es necesario ponerlo)
    - El **único parámetro** que tenemos que poner es el **segundo argumento**
  - Por ejemplo, para el **operador de suma**

```
tipo operator+(const tipo &b)
```

  - Como el método anterior se implementa como función miembro
    - No ponemos la referencia a *this* porque es implícita
  - **Se retorna una referencia al tipo** para que puedan **encadenarse sumas** de la forma: d = a + b + c

Deusto

SOBRECARGA COMO MÉTODO (2)

Point

```
class Point
{
    int x;
    int y;

public:
    Point(int x, int y):x(x), y(y) {}

    Point operator+(const Point &b) const
    {
        Point p = *this;
        p.x += b.x;
        p.y += b.y;
        return p;
    }
};
```

Main

```
int main()
{
    Point p1(2,3);
    Point p2(4,7);
    Point p3 = p1 + p2;
    cout << p3.x << " " << p3.y << endl;
    return 0;
}
```

Deusto

SOBRECARGA DE OPERADOR +

- El operador suma por convenio no modifica el objeto sobre el que se aplica
  - Por ello hemos indicado que el método es **const**
- Lo que hacemos es **devolver un nuevo objeto** con el resultado de la operación
  - Al **encadenar operaciones** dichos objetos se van pasando a los distintos operados sobrecargados

```
Point operator+(const Point &b) const
{
    Point p = *this;
    p.x += b.x;
    p.y += b.y;
    return p;
}
```

Deusto

SOBRECARGA DE OPERADOR += (1)

- En este caso el **operador +=** debe **modificar el objeto** sobre el que se aplica >> this
  - Por ello, el método **no es const**
  - La modificación del valor **se aplica sobre el propio objeto this**
  - Se **retorna el propio objeto this**

```
Point& operator+=(const Point &b)
{
    this->x += b.x;
    this->y += b.y;
    return *this;
}
```

- Cuidado, en este caso devolvemos una referencia al objeto this. Sería incorrecto devolver una referencia a un objeto creado en el método.
  - El ámbito del objeto local terminaría al finalizar el método >> Error.

Deusto

SOBRECARGA DE OPERADOR += (2)

Point

```
class Point
{
    int x;
    int y;
public:
    Point(int x, int y):x(x), y(y) {}

     Point& operator+=(const Point &b)
     {
          this->x += b.x;
          this->y += b.y;
          return *this;
     }
};
```

Main

```
int main()
{
    Point p1(2,3);
    Point p2(4,7);
    p1 += p2;
    cout << p1.x << " " << p1.y << endl;
    return 0;
}
```

Deusto

PARÁMETROS DE OTROS TIPOS

- Podríamos **sobrecargar el mismo operador varias veces**
  - Deberían **cambiar los parámetros en cada sobrecarga** para poder distinguirlos
    - Un primer parámetro implícito >> this
    - Un segundo parámetro de otro tipo

```
Point& operator+=(int a)
{
    this->x += a;
   this->y += a;
   return *this;
}
```

Deusto

OPERADOR INCREMENTO

- Pre-incremento: **++a**

```
Point& operator++()
{
    this->x++; //modificamos el valor y lo devolvemos
    this->y++;
    return *this;
}
```

- Post-incremento: **a++**

```
Point operator++(int) // hay que poner int para diferenciar del primero. Sin uso
{
    Point p = *this; //hay que hacer una copia porque es post-incremento
    this->x++;
    this->y++;
    return p; //devolvemos el valor original antes de modificar
}
```

Deusto

SOBRECARGA DE OPERADORES E/S (1)

Salida Estándar

- En C++ es posible indicar **cómo van a llevarse a cabo las operaciones de entrada/salida de un objeto** con **<iostream>**
  - Se realiza mediante la **sobrecarga de los operadores << y >>**
  - La idea es poder hacer operaciones tales como:

```
Point p;
cout << p << endl; //escribe el punto p a la salida estándar
cin >> p; //lee un punto a p desde la entrada estándar.
```

  - En este caso, como su primer parámetro no es el objeto Point, sino cout/cin, **no se pueden implementar como funciones miembro** de la clase
    - Hay que implementarlos como **funciones externas**

Deusto

SOBRECARGA DE OPERADORES E/S (2)

Salida Estándar

- La definición de **operator<<** es:

```
ostream& operator<< (ostream &out, const tipo &t);
```

- Tiene que **recibir dos parámetros**: uno de tipo **ostream&** y otro del **tipo concreto a escribir** en la salida estándar
- Cuando hacemos **cout << a;**
  - **cout** es el primer parámetro de la función
  - **a** es del tipo para el que hemos sobrecargado el operador
- El **retorno de cout** permite **encadenar >>** **cout << p1 << "" << p2;**
- Si implementamos como función miembro, se asimiría que es necesario un primer parámetro extra (this) → no funcionaría

EJEMPLO OPERATOR<<

Point

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

class Point
{
   float x;
   float y;

public:
   Point(float x, float y): x(x), y(y) {}

   float getX() const {
      return x;
   }

   float getY() const {
      return y;
   }
};
```

Main

```
ostream& operator<<(ostream &out, const Point &p)
{
   out << "(" << p.getX() << "," << p.getY() << ")";
   return out;
}

int main(void)
{
   Point *p = new Point(5, 7);
   cout << *p << endl;
   delete p;
}
```

Deusto

SOBRECARGA DE OPERADORES E/S (3)

Entrada Estándar

- La librería **<iostream>** permite leer de la entrada estándar de una forma sencilla
  - Además de cout → salida estándar
  - Proporciona **cin** → **entrada estándar**
- Para leer desde cin utilizamos el **operador >>**
  - Sabe **leer tipos simples** de C++ (int, float, char, boolean)
- Por ejemplo:

```
#include <iostream>
using namespace std;
. . .
int a;
cout << "Escribe un entero: ";
cin >>  a; //lee un entero de la consola
```

Deusto

SOBRECARGA DE OPERADORES E/S (4)

Entrada Estándar

- Podemos hacer lo mismo para leer **tipos complejos**
- Tenemos que **sobrecargar el operator>>**
- La definición del operador >> es:

```
istream& operator>> (istream& is, tipo& t);
```

- Podremos hacer **cin >> a;**
  - Donde **cin** representa la entrada
  - Donde **a** es el objeto donde leer
- En este caso el objeto recibido **no puede ser const**
  - **Necesitamos modificarlo** para leer en él los datos

Deusto

EJEMPLO OPERATOR>>

Point

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

class Point
{
   float x, y;
public:
   Point() {}
   Point(float x, float y): x(x), y(y) {}
   float getX() const { return x; }
   float getY() const { return y; }
   float setX(float x) { this->x = x; }
   float setY(float y) { this->y = y; }
};
```

Main

```
ostream& operator<<(ostream &out, const Point &p)
{
   out << "(" << p.getX() << ", " << p.getY() << ")";
   return out;
}
istream& operator>>(istream &in, Point &p)
{
    int x,y;
    cout << "Escriba un valor para X: ";
    cin >> x;
    cout << "Escriba un valor para Y: ";
    cin >> y;
    p.setX(x);
    p.setY(y);
    return in;
}
int main(void)
{
   Point *p = new Point();
   cin >> *p;
   cout << *p << endl;
   delete p;
}
```

Deusto

FUNCIONES FRIEND

- En los ejemplos anteriores hemos necesitado **funciones get/set**
- ¿Podríamos **leer/escribir** un objeto de una clase **sin dichos métodos**?
  - Desde fuera de la clase, en principio, no
- Pero las funciones **operator<< y operator>> ni pueden ser funciones miembro, ni pueden acceder sin get/set**
  - ¿Qué podemos hacer?
- Sin embargo, C++ proporciona la posibilidad de **saltarse la encapsulación**
  - Se lleva a cabo mediante la definición de **funciones friend**
  - Una función friend de una clase
    - **Puede acceder a las variables: private, protected**
    - **No es miembro de la clase** >> es especial
  - Lo mejor es **intentar no utilizarlas** >> pero puede ser necesario

Deusto

OPERATOR<< COMO FRIEND

Point

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

class Point
{
    float x;
    float y;
public:
    Point() {}
    Point(float x, float y): x(x), y(y) {}
    friend ostream& operator<<(ostream &out, const Point &p); //función amiga
};
```

Main

```
ostream& operator<<(ostream &out, const Point &p)
{
    out << "(" << p.x << ", " << p.y << ")"; //Podemos acceder sin get
    return out;
}
int main(void)
{
    Point *p = new Point(5, 7);
    cout << *p << endl;
    delete p;
}
```

Deusto

FUNCIONES FRIEND

- En los ejemplos anteriores hemos necesitado **funciones get/set**
- ¿Podríamos **leer/escribir** un objeto de una clase **sin dichos métodos**?
  - Desde fuera de la clase, en principio, no
- Pero las funciones **operator<< y operator>> ni pueden ser funciones miembro, ni pueden acceder sin get/set**
  - ¿Qué podemos hacer?
- Sin embargo, C++ proporciona la posibilidad de **saltarse la encapsulación**
  - Se lleva a cabo mediante la definición de **funciones friend**
  - Una función friend de una clase
    - **Puede acceder a las variables: private, protected**
    - **No es miembro de la clase** >> es especial
  - Lo mejor es **intentar no utilizarlas** >> pero puede ser necesario

Deusto

SOBRECARGA DE OPERADORES E/S (5)

- Lo interesante es que ***ostream*** e ***istream***
  - Representan **objetos de una clase genérica**
  - Pueden estar implementados como:
    - **Consola**
    - Salida a **fichero**
    - **Impresora**
    - **Red**
- Conocer a fondo la **librería <iostream>** permite realizar operaciones de E/S en C++ de una forma más sencilla
  - **Reutilizar código**
  - **Cambiar fácilmente la salida/entrada de un programa**

Deusto

SOBRECARGA OPERADOR =

- **C++ no sabe cómo asignar los objetos entre sí.** Únicamente copia los tipos básicos >> **Problemas con arrays, punteros a otros objetos, etc.**
- Es el mismo problema que el constructor copia visto anteriormente.
- Es necesario recordar que las siguientes soluciones son distintas:
  - C++ llama a un constructor copia (si existe) cuando:

```
Alumno b(a); // Lo usamos explícitamente
Alumno b = a; // Construcción y asignar en la misma línea
```

  - Sin embargo, en el siguiente caso C++ llama al operador de asignación para dicha clase (si existe):

```
Alumno b; // Primero se crea el objeto de alguna forma
[…]
b = a; // Posteriormente se le asigna otro objeto
```

Deusto

EJERCICIOS

Programación en C++: Enunciados

- Ej09: Polimorfismo y sobrecarga
- Ej10: Sobrecarga de operadores
- Ej11: Sobrecarga de operadores