Deusto

PROGRAMACIÓN EN C++

ENUNCIADOS DE EJERCICIOS

Asignatura

PROGRAMACIÓN IV

Deusto

Ej01: CLASES

(Pr_Cpp.Ej01)

- Implementar y probar la siguiente clase en C++
  - Define la funcionalidad de un *array* de tamaño variable
  - Declararla dentro del *namespace* **containers**

```
class ArrayInt
{
    private:
        unsigned int capacity;
        int *array;

    public:
        ArrayInt();
        ArrayInt(unsigned int capacity);
        ~ArrayInt();

        void setValue(unsigned int index, int data);
        int getValue(unsigned int index);
        void setCapacity(unsigned int capacity);
        unsigned int getCapacity();
};
```

Deusto

Ej02: CLASES (1)

(Pr_Cpp.Ej02)

- Implementar y probar la siguiente clase en C++
  - Define la funcionalidad de una *pila* de elementos
  - Declararla dentro del *namespace* **containers**

```
class Element
{
private:
    int data;
    Element * next;

public:
    Element(int data);

    int getData();
    void setNext(Element *e);
    Element* getNext() const;
};
```

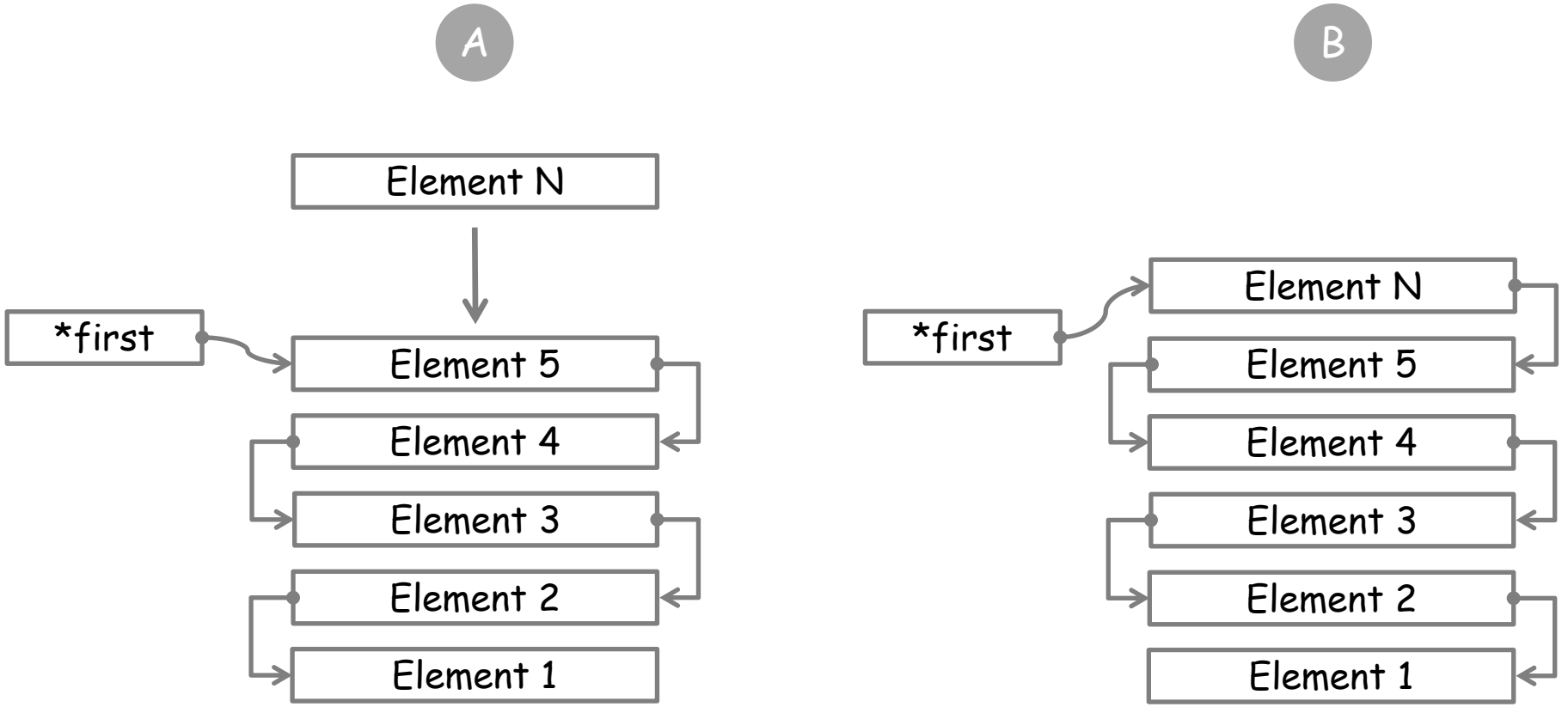
```
class Stack
{
private:
    Element *first;
    unsigned int size;

public:
    Stack();
    ~Stack();

    void push(int dato);
    int pop();
    int poll();
    void clear();
    unsigned int getSize();
};
```

Deusto

Ej02: CLASES (2)

(Pr_Cpp.Ej02)

- Representación del funcionamiento de la pila



Deusto

Ej03: REFERENCIAS (1)

(Pr_Cpp.Ej03)

- Implementar y probar la siguiente clase

```
class Point
{
private:
    float x, y;

public:
    Point(int x, int y);

    void suma1(Point p);
    void suma2(Point &p);
    void suma3(Point *p);
    Point getSuma(Point &p);
    void print();
};
```

Deusto

Ej03: REFERENCIAS (2)

(Pr_Cpp.Ej03)

- Implementar tres métodos **swap** que *intenten* intercambiar los valores de los puntos:

```
void swap1(Point p);
void swap2(Point &p);
void swap3(Point *p);
```

- Visualizar los valores del punto pasado como parámetro para comprobar en qué casos han cambiado.

Deusto

Ej04: CONST

(Pr_Cpp.Ej04)

- Implementar y probar la siguiente clase con métodos y argumentos constantes.
- Plantear situaciones en las que se intenten modificar sus valores.

```
class Point
{
private:
    float x, y;

public:
    Point(int x, int y);

    void setX(float x);
    void setY(float y);

    float getX() const;
    float getY() const;

    void sumar(const Point &p);
};
```

Deusto

Ej05: CONSTRUCTOR COPIA (1)

(Pr_Cpp.Ej05)

- Definir e implementar la siguiente clase

```
class Alumno
{
private:
    static int counter;
    int id;
    char *nombre;

public:
    Alumno(const char *nombre);
    ~Alumno();

    int getID();
    char *getNombre();
};
```

- **Imprimir cualquier cosa por pantalla en el constructor y el destructor** para ver cuándo son llamados dichos métodos al crear y eliminar objetos

Deusto

Ej05: CONSTRUCTOR COPIA (2)

(Pr_Cpp.Ej05)

- Implementar un **programa principal** que construya objetos Alumno

```
Alumno a("Juan"); //Alumno original
Alumno b = a; //Debe ser una copia de a
Alumno c(b); //Debe ser una copia de b
Alumno *d = new Alumno(c); // Copia de c. delete necesario
```

- Y utilice las siguientes funciones para imprimir todos los alumnos

```
void printAlumno(Alumno alumno)
{
   // Imprime el ID y el nombre del Alumno
}
void printAlumnoRef(Alumno &alumno)
{
   //Imprime el ID y el nombre del Alumno
}
void printAlumnoP(Alumno * alumno)
{
   //Imprime el ID y el nombre del Alumno
}
```

Deusto

Ej05: CONSTRUCTOR COPIA (3)

(Pr_Cpp.Ej05)

- El ejemplo anterior es incorrecto ya que no hemos definido cómo se copian los objetos Alumno
  - Tienen datos internos que deben ser copiados >> NO únicamente los punteros de los nombres
  - Se han creado por error datos compartidos que al ser eliminados por los destructores producen errores.
- Implementar el **constructor copia** de la clase **Alumno**

```
Alumno(const Alumno &a);
```

- Imprimir algo en ese constructor para ver cuándo es llamado

Deusto

Ej05: CONSTRUCTOR COPIA (4)

(Pr_Cpp.Ej05)

- Una vez implementado el ejercicio, si suponemos el siguiente código para el programa principal:

```
int main()
{
    Alumno a("Juan");
    printAlumnoRef(a);
    printAlumno(a);
    Alumno b = a;
    Alumno c(b);
    printAlumnoP(&c);
    Alumno *d = new Alumno(c);
    printAlumnoP(d);
    printAlumnoRef(a);
    delete d;

    return 0;
}
```

¿Cuantas veces se invoca?

- Constructor ?
- Constructor copia ?
- Destructor ?

¿En qué lugares se hace?

Deusto

Ej06: HERENCIA

(Pr_Cpp.Ej06-07)

- Programar utilizando herencia en C++ (teniendo en cuenta los constructores y destructores)
  - Clase **Persona**
    - int edad >> getEdad()
    - char* nombre >> getNombre()
  - Clase **Alumno** (hereda de Persona)
    - int numAsignaturas >> getNumAsignaturas()
    - float notas >> es un array de numAsignaturas
  - Crear funciones en el **programa principal** que reciban un **Alumno/Persona** e impriman
    - La información de una Persona
    - Las notas de un Alumno

Deusto

Ej07: POLIMORFISMO

(Pr_Cpp.Ej06-07)

- En el ejemplo anterior añadir un método **diHola()** a la jerarquía de clases
  - Implementarlo de forma distinta en **Persona** y **Alumno**
- Utilizar punteros de distintos niveles de la jerarquía para poder utilizar el concepto de polimorfismo. Por ejemplo:

```
Alumno *a = new Alumno(...);
Persona *p = a;

p->diHola();
a->diHola();
```

- Comprobar si se llama de forma correcta y si es necesario realizar alguna modificación en el código para que funcione de la forma esperada (polimorfismo)

Deusto

Ej08: CLASES ABSTRACTAS

(Pr_Cpp.Ej08)

- Hacer la clase **Persona** *abstracta* definiendo el método:

```
escribirEnFichero(const char* fichero);
```

- La clase **Alumno** debe implementar la funcionalidad para escribir algo en pantalla al llamar al método.
- Comprobar que no se pueden crear instancias de la clase Persona
  - Sí se puede utilizar el tipo Persona para referenciar a subtipos de la jerarquía (por ejemplo, Alumno).

Deusto

Ej09: POLIMORFISMO Y SOBRECARGA (INICIACIÓN)

(Pr_Cpp.Ej09)

- Implementar una clase **Punto** y una clase hija **Punto3D** con sus constructores correspondientes para poder inicializarlos
- Implementar un **método polimórfico** que visualice los valores de los puntos
- Crear un **array** capaz de contener 5 (punteros a) puntos de cualquier tipo
- **Recorrer el array** y solicitar a los puntos que muestren por pantalla sus valores
- Implementar una función sobre Punto3D que **sobrecargue el operador ***
  - Debe estar sobrecargado el operador para permitir multiplicar un punto por otro y devolver un tercer punto con el resultado. Y debe estar sobrecargado para escalar un punto una cantidad entera, modificando el punto escalado.

Deusto

Ej10: SOBRECARGA DE OPERADORES

(Pr_Cpp.Ej10)

- Crear la clase **Point3D** y definir las siguientes operaciones mediante sobrecarga de operadores:
  - Sumar dos puntos >> p1 + p2 >> (x1 + x2, y1 + y2, z1 + z2)
  - Sumar y asignar un punto a otro >> p1 += p2
  - Producto escalar de dos puntos >> p1 * p2 >> x1*x2 + y1*y2 + z1*z2
  - Producto de un vector por un escalar >> p1 * s >> (x*s, y*s, z*s)
  - Leer un punto de la entrada estándar
  - Imprimir un punto a la entrada estándar
- Realizar un programa principal de prueba que permita probar la funcionalidad de la clase

Deusto

Ej11: SOBRECARGA DE OPERADORES

(Pr_Cpp.Ej11)

- Modificar el Ej05 para crear un objeto mediante una asignación

```
Alumno e("Alumno E");

e = a;    // donde a es otro Alumno ya creado
```

- Comprobar que esto no es correcto, ya que la asignación por defecto únicamente copia los punteros pero no el contenido >> Produce errores en la memoria copiada
- Resolver el problema implementando (sobrecargando) el **operador de asignación** para la clase **Alumno** con la finalidad de realizar la copia de los datos internos de la forma correcta

```
Alumno& operator=(const Alumno &a);
```