



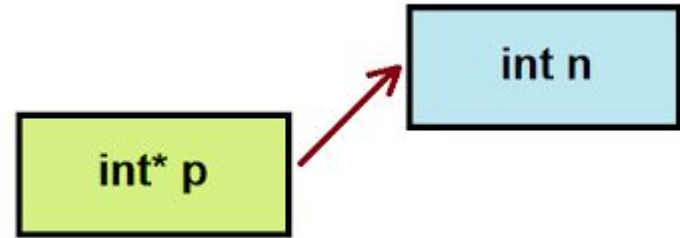
Universidad de Jaén

# Gestión de memoria y definición de patrones

## Lección 3:

- Gestión de memoria
- Definición y uso de patrones
- Definición y uso de operadores

## Estructuras de Datos



# Parte I:

## Gestión de memoria

- Tipos de asignación de memoria
- Problemas de memoria
- Punteros dobles
- Herramientas de monitorización de memoria

# Motivación

- Antes de entrar en la materia de la asignatura, repasaremos un aspecto fundamental como es el manejo de memoria
- También estudiaremos dos elementos de C++ (y otros lenguajes OO) muy útiles para la implementación de EEDD:
  - El soporte de patrones
  - El soporte de operadores

# Gestión de memoria

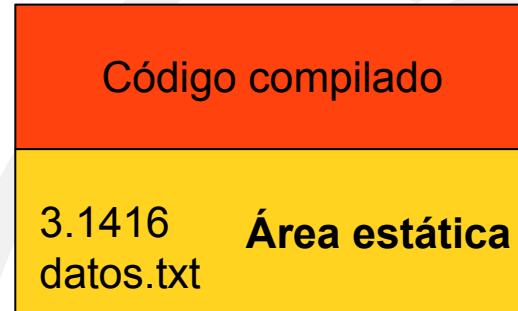
- En memoria primaria existen tres tipos de asignación de memoria:
  - Estática
  - Automática (stack)
  - Dinámica (heap)

# Asignación estática

- Se realiza únicamente en tiempo de compilación
- Sirve para guardar variables globales y constantes
- De escasa utilidad en la implementación de EEDD

```
const float PI = 3.1416  
char fichero[] = "datos.txt"
```

compilación



# Asignación automática

- Durante la ejecución cuando se entra en un bloque de código con variables/objetos locales
- Se realiza en la **pila de la aplicación**
- Al salir de la función, el espacio es liberado automáticamente

```
void f(int k) {  
    int p;  
    float arr[3];  
    ...  
}
```

durante llamada a f()



Código compilado

Área estática

arr[0]

arr[1]

arr[2]

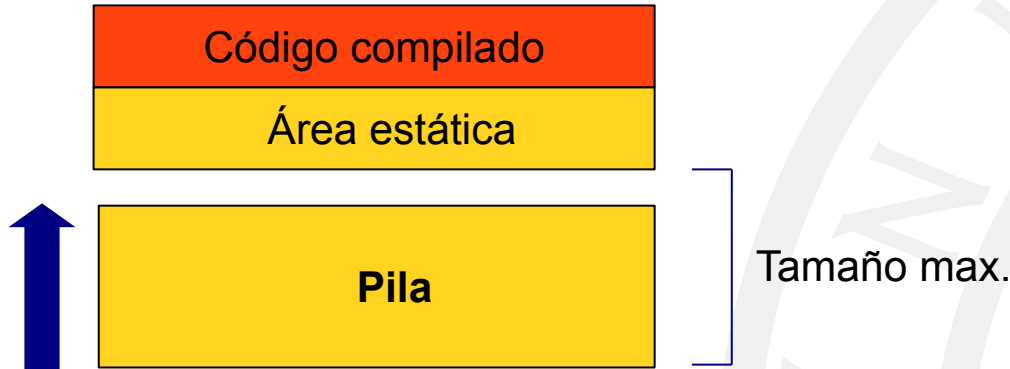
p

k

**Pila**

# Gestión de la pila

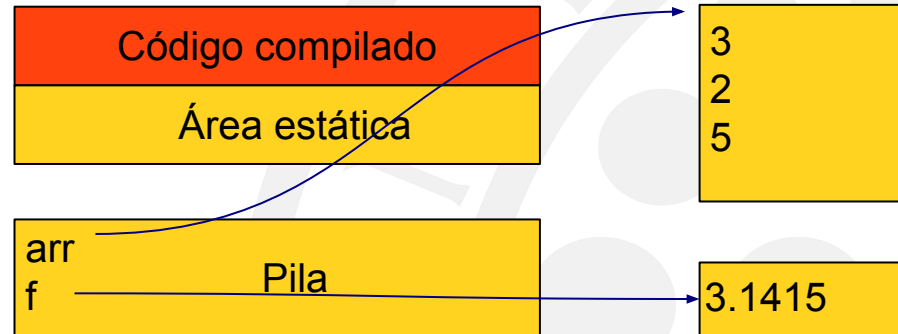
- La pila tiene un tamaño fijo (parámetro de compilación)
- Crece en dirección hacia el código compilado cuando se asigna espacio y decrece en dirección contraria al liberar
- La asignación en pila es rápida y sencilla



# Asignación dinámica

- Tiene lugar en el **heap** o zona libre
- Virtualmente puede utilizar toda la memoria disponible en la máquina
- Se maneja mediante punteros y la asignación/ liberación se hace de forma explícita en tiempo de ejecución

```
float *f = new float  
int *arr = new int[3]  
  
*f = 3.1415  
arr[0] = 3  
arr[1] = 2  
arr[2] = 5  
...
```





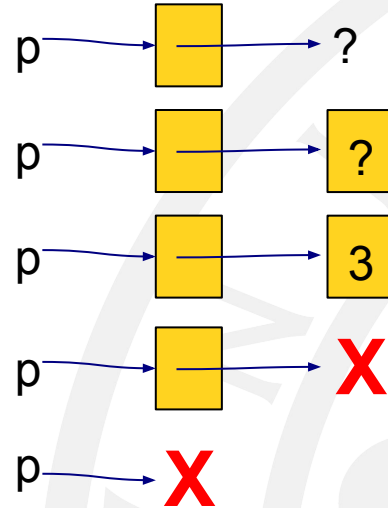
## EEDD y memoria dinámica

- La memoria dinámica es la más abundante y la que permite un uso más flexible
- Inconveniente: asignación/liberación mucho más lenta que en la pila de la aplicación
- Es la que utilizaremos preferentemente en la implementación de EEDD
- Por tanto hay que manejar a la perfección la asignación/uso/liberación de memoria dinámica mediante punteros

# Punteros dobles

- Una técnica que usaremos frecuentemente y que hay que dominar es el uso de punteros dobles

```
int **p = new int*;  
  
*p = new int;  
  
**p = 3;  
  
delete *p;  
  
delete p;
```



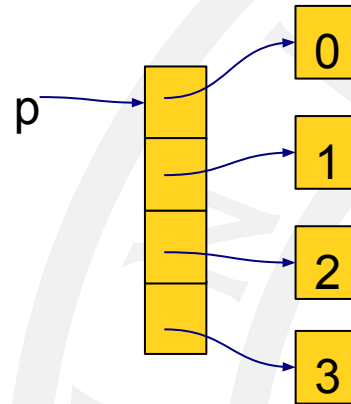
# Punteros a vectores de punteros

- También es frecuente el uso de punteros dobles para el manejo de vectores de punteros a datos/objetos

```
// Creación
int **p = new int*[4];

for (int c = 0; c < 4; c++) {
    p[c] = new int;
    *p[c] = c;
}

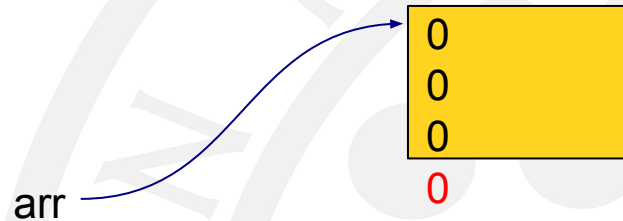
...
// Destrucción
for (int c = 0; c < 4; c++)
    delete p[c]
delete[] p;
```



# Desbordamientos (heap overflows)

- El manejo de la memoria dinámica es delicado y es fácil cometer errores con consecuencias imprevisibles
- Un error frecuente son los desbordamientos o heap overflows (acceder fuera de la memoria asignada)

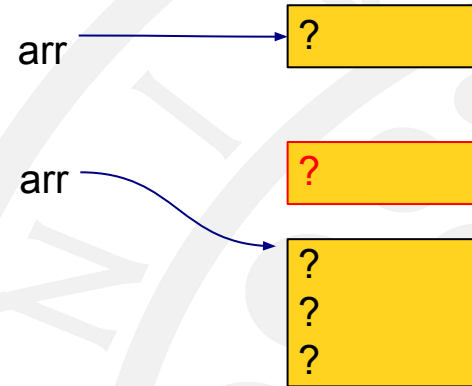
```
int *arr = new int[3];  
  
for (int p = 0; p <= 3; p++)  
    arr[p] = 0  
  
delete[] arr;
```



# Fugas de memoria (memory leaks)

- Igual de común aunque algo menos grave son la fugas de memoria o memory leaks
- Ocurre siempre que se reserva memoria y por descuido no se libera

```
int *arr = new int;  
...  
// Memory leak! El entero reservado  
// inicialmente se pierde al  
// reasignar el puntero  
arr = new int[3];  
  
delete[] arr;
```



## Más problemas...

- Una asignación de memoria puede fracasar si no hay memoria libre suficiente. En ese caso el operador new lanza una excepción **bad\_alloc**
- Nunca acceder mediante punteros sin inicializar o apuntando a nulo
- Las dobles liberaciones también suelen dar problemas

```
try {  
    int *arr =  
        new int[100000000000000];  
}  
catch(bad_alloc e) {  
    // Asignación imposible!  
}
```

```
int *arr;  
  
*arr = 3; // Error!  
  
arr = 0  
*arr = 3; // Error!
```

```
int *arr = new int;  
  
delete arr;  
  
// Ya liberado!  
delete arr;
```

## Cómo evitar problemas de memoria

- Revisar que los accesos a los arrays no exceden sus límites
- Asegurar que cada asignación con *new* tiene su correspondiente *delete*. Liberar siempre los arrays con *delete[]*
- Implementar destructor, y evaluar la necesidad de constructor copia y operador de asignación si una clase maneja memoria dinámica
- No acceder a punteros no iniciados o apuntando a *NULL*, y evitar dobles liberaciones

# Herramientas de monitorización de memoria

- Herramientas comerciales y gratuitas que permiten detectar problemas de memoria:

- Valgrind
- Memwatch
- Insure++
- etc.

```
==2009== 512 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2009==    at 0x400483E4: malloc (vg_clientfuncs.c:100)
==2009==    by 0x80484D4: main (in /jfs/article/sample1)
==2009==    by 0x40271507: __libc_start_main
    (../sysdeps/generic/libc-start.c:129)
==2009==    by 0x80483B1: free@@GLIBC_2.0 (in /jfs/article/sample1)
==2009==
==2009== 5120 bytes in 512 blocks are definitely lost in loss record 2 of 2
==2009==    at 0x400483E4: malloc (vg_clientfuncs.c:100)
==2009==    by 0x80484A0: get_mem (in /jfs/article/sample1)
==2009==    by 0x8048519: main (in /jfs/article/sample1)
==2009==    by 0x40271507: __libc_start_main
    (../sysdeps/generic/libc-start.c:129)
==2009==
==2009== LEAK SUMMARY:
==2009==    definitely lost: 5632 bytes in 513 blocks.
```



## Parte II:

# Definición y uso de patrones

- Definición
- Utilización de patrones
- Implementación de patrones
- Ventajas de los patrones

# Definición de patrones

- Hay ciertas clases que por su naturaleza, deben ser parametrizables, de manera similar a las funciones
- La mayoría de clases que implementan EEDD pertenecen a esta categoría
- El soporte de patrones de C++ nos va ayudar a conseguir implementar este tipo de clases

# Un vector de enteros en memoria dinámica

Vamos a implementar una clase que encapsule un vector de enteros en memoria dinámica

```
class VectorInt {  
    int *mem;  
    long int tam;  
  
public:  
    VectorInt(long int atam) { mem = new int[tam = atam]; }  
  
    ~VectorInt() { delete[] mem; }  
  
    int leer(long int pos) { return mem[pos]; }  
    void escribir(long int pos, const int &valor)  
        { mem[pos] = valor; }  
};
```

## Un vector de enteros en memoria dinámica

Al destruirse el objeto, éste eliminaría automáticamente la memoria dinámica asignada

```
VectorInt arr(100);  
  
for (int c = 0; c < 100; c++)  
    arr.escribir(c, 0);  
  
VectorInt arr2(50);  
  
for (int c = 0; c < 50; c++)  
    arr2.escribir(c, arr.leer(c));
```

## Un vector de flotantes

Si queremos un vector para flotantes, hay que crear una nueva clase a partir de la original realizando las modificaciones con cuidado... y así con cada tipo que queramos utilizar.

```
class VectorFloat {
    float *mem;
    long int tam;

public:
    VectorFloat(long int atam) { mem = new float[tam = atam]; }

    ~VectorFloat() { delete[] mem; }

    float leer(long int pos) { return mem[pos]; }
    void escribir(long int pos, const float &valor)
        { mem[pos] = valor; }
};
```

## Vectores sin tipo

- Para evitar tener que definir una clase distinta para cada tipo existen varias soluciones
- La primera es no utilizar tipos en absoluto

```
class Vector {  
    void *mem;  
    long int tam;  
    long int tamdato;  
  
public:  
    Vector(long int atam, long int atamdato);  
  
    ~Vector() { delete[] mem; }  
  
    void leer(long int pos, void *dato) {  
        memcpy(dato, mem + tamdato * pos, tamdato);  
    }  
}
```

## Vectores sin tipo (cont.)

```
void escribir(long int pos, const void *dato) {  
    memcpy(mem + tamdato * pos, dato, tamdato);  
}  
};  
  
Vector::Vector(long int atam, long int atamdato)  
{  
    tam = atam;  
    tamdato = atamdato;  
    mem = new char[tamdato * tam];  
}
```

## Vectores sin tipo (cont.)

- Implementar y utilizar este tipo de clases es tedioso y propenso a errores

```
int main() {  
    Vector ai(10, sizeof(int));  
    Vector af(25, sizeof(float));  
    int itmp = 5;  
    float ftmp = 2.3;  
  
    ai.escribir(3, &itmp);  
    af.escribir(7, &ftmp);  
  
    ai.leer(3, &itmp);  
    af.leer(7, &ftmp);  
    return 0;  
}
```



# Vectores polimorfos

- Más elegante es conseguir genericidad a partir de polimorfismo

```
class Dato {};  
  
class Vector {  
    Dato **mem;  
    long int tam;  
  
public:  
    Vector(long int atam) { mem = new Dato*[tam = atam]; }  
  
    ~Vector() { delete[] mem; }  
  
    Dato* leer(long int pos) { return mem[pos]; }  
    void escribir(long int pos, const Dato *valor)  
        { mem[pos] = valor; }  
};
```

## Vectores polimorfos (cont.)

- C++ no incluye una clase padre universal, así que obliga a heredar siempre de la clase utilizada en el vector polimorfo
- No funciona con tipos simples (int, float, etc.)


```
class MiDato : public Dato {};  
  
int main() {  
    Vector arr(10);  
    MiDato *d = new MiDato();  
    ...  
    arr.escribir(5, d);  
    ...  
    d = (MiDato *) arr.leer(7); // Casting necesario  
};
```

# Patrones

- C++ incluye un mecanismo mucho mejor para generar clases parametrizadas: **los patrones**
- Un patrón es similar a una clase, aunque incluye uno o más tipos como parámetros que son utilizados en su implementación
- El tipo puede ser una clase, una estructura o cualquier tipo simple

## Patrones (cont.)

Vamos a convertir el vector en un patrón que para poder utilizarlo con los tipos que necesitamos

```
template<class T>  Parámetro de tipo
class Vector {
    T *mem;
    long int tam;

public:
    Vector(long int atam) { mem = new T[tam = atam]; }

    ~Vector() { delete[] mem; }

    T leer(long int pos) { return mem[pos]; }
    void escribir(long int pos, const T &valor)
        { mem[pos] = valor; }


};
```

# Utilización de patrones

- La instanciación del patrón en clase (indicando el tipo del parámetro) y instanciación de la clase en objeto se hace al mismo tiempo

```
int main() {  
    Vector<int> ai(10);  
    Vector<float> af(25);  
  
    ai.escribir(3, 5);  
    af.escribir(7, 2.3);  
  
    cout << ai.leer(3) << " " << af.leer(7) << endl;  
    return 0;  
}
```

Parámetro T



# Implementación de patrones

- Los patrones deben implementarse íntegramente en el fichero de cabecera (el fichero .cpp normalmente no existe)
- Las operaciones se implementarán a continuación de la declaración de la clase, en el mismo fichero de cabecera
- Puede hacerse cualquier suposición sobre el tipo del patrón (p. ej.: que puede ser comparado con <)

# Implementación de patrones

Vamos a implementar el constructor del patrón tras la definición de la clase (operación no inline)

```
template<class T>
class Vector {
    T *mem;
    long int tam;

public:
    Vector(long int atam);
    // ... resto de operaciones
};

template<class T>
Vector<T>::Vector(long int atam) {
    mem = new T[tam = atam];
}
```

## Ventajas de los patrones

- Mecanismo sencillo y potente
- Una vez instanciado el patrón, hay comprobación estricta de tipos
- No impone en principio ninguna condición sobre el tipo del patrón



## Parte III:

# Definición y uso de operadores

- Implementación de operadores
- Operador [], =, ==
- Utilización de operadores

# Implementación de operadores

- No aportan una funcionalidad nueva a una clase
- Un operador puede ser más cómodo e intuitivo que una operación ordinaria
- Operadores unarios:

```
++a → A::operator++()  
a++ → A::operator++(int)  
*a → A::operator*()
```

- Operadores binarios:

```
a+=b → A& A::operator+=(B &b)  
a+b → A A::operator+(B &b)  
a<b → bool A::operator<(B &b)
```

# El operador []

Vamos a definir los operadores [], = y == sobre nuestro vector

```
template<class T>
class Vector {
    T *mem;
    long int tam;

public:
    Vector(long int atam) { mem = new T[tam = atam]; }

    ~Vector() { delete[] mem; }

    // Este operador sustituye a las dos operaciones de lectura/escritura
    T &operator[](int pos) { return mem[pos]; }
    bool operator==(Vector &arr);
    Vector &operator=(Vector &arr);
};
```

## El operador=

- El operador= es más importante que los demás y cumple la función especial de asegurar una copia correcta de los objetos

```
template<class T>
Vector& Vector<T>::operator=(Vector &arr) {
    if (&arr != this){
        delete[] mem;
        tam = arr.tam;

        mem = new T[tam];
        for (int c = 0; c < tam; c++) {
            mem[c] = arr.mem[c];
        }
        return *this;
    }
}
```

## El operador==

Esta es la implementación del operador de comparación. Es poco común su definición en un vector

```
template<class T>
bool Vector<T>::operator==(Vector &arr) {
    if (tam != arr.tam)
        return false;

    for (int c = 0; c < tam; c++) {
        if (mem[c] != arr.mem[c])
            return false;
    }
    return true;
}
```

# Utilización de operadores

Con los operadores, el vector puede ser manejado de forma muy natural y compacta

```
int main() {  
    Vector<int> ai(10);  
    Vector<int> ai2(25);  
  
    ai[3] = 5  
    ai2[7] = 2  
  
    cout << ai[3] << " " << ai[7] << endl;  
    ai = ai2;  
    if (ai == ai2)  
        cout << "los vectores son iguales" << endl;  
    return 0;  
}
```

# Conclusiones

- Es importante diseñar la organización de los datos en memoria antes de realizar la implementación, es decir, trabajar el concepto de “gestión de memoria”
- Implementar las EEDD mediante patrones/plantillas de clase para poder instanciarlas a cualquier tipo de objeto
- Los operadores son muy útiles en C++ y se hace uso extensivo de ellos en la librería STL
- Siempre que se diseñe una clase hay que plantearse si es o no imprescindible el *operator=* y el constructor copia

## De ahora en adelante...

- Veremos la implementación de EEDD dinámicas, esto es, que crecen/decrecen en tiempo de ejecución
- La primera de ellas es el vector dinámico

## Lección 4: Vectores dinámicos