

Informática Industrial

El patrón RAI

Del C al C++

- ▶ El patrón **RAII** (“*resource acquisition is initialization*”: la adquisición del recurso es la inicialización), se utiliza para administrar de forma segura la adquisición y posterior devolución de un recurso de programación:
- ▶ Ejemplos de recursos:
 - ▶ Conexión a bases de datos remotas
 - ▶ Apertura y posterior cierre de ficheros.
 - ▶ Adquisición y devolución de memoria desde el *heap*: *smart pointers* (punteros inteligentes).
 - ▶ Trabajo seguro con *mutexes*



Del C al C++

- ▶ El patrón **RAII** básicamente consiste en:
 - ▶ Encapsular el proceso de adquisición, uso y liberación del recurso en una clase diseñada para ese propósito.
 - ▶ Adquirir el recurso en el constructor de la clase
 - ▶ Liberar el recurso en el destructor de la clase
 - ▶ **Inicializar** un objeto de esa clase en la función que lo vaya a utilizar
 - ▶ **Utilizar** el objeto de la manera apropiada.
 - ▶ Cuando la ejecución de la función termina, los objetos locales (con espacio de memoria reservado en el *stack*) serán destruidos en el orden inverso a su declaración. En el **destructor del objeto se libera el recurso adquirido.**
 - ▶ Este mecanismo funciona aún en presencia de **excepciones.**



Del C al C++

- ▶ Las clases contenedores (vector, lista, etc) que pueden crecer tanto como se quiera (mientras haya memoria), en su diseño, reservan espacio en el *heap* y liberan ese espacio cuando abandonan el alcance de la función en que fueron definidas. O sea, implementan, tras bambalinas, el patrón RAII:

```
#include <iostream>

using namespace std;

void foo ()
{
    vector <double> v (10000);

    /*trabajamos con vector*/
}

int main ()
{
    foo ();
}
```

La clase **vector** se crea en *stack* de **foo**, pero los elementos del vector se reservan en el *heap*.

Cuando se llega la final de **foo** se destruyen todos los objetos locales y como parte del **destructor** de **v** se libera la memoria usando **delete []...**
¡Pero no tenemos que preocuparnos!

Del C al C++

```
#include <iostream>
#include <fstream>
#include <exception>
#include <stdexcept>

class fichero
{
public:
    explicit fichero(std::string nombre, std::string texto)
    {
        fich.open (nombre, std::ofstream::out | std::ofstream::app);
        if (fich.is_open())
            fich << texto;
        std::cout << "abriendo fichero: " << nombre << std::endl;
    }
    ~fichero()
    {
        if (fich.is_open())
        {
            fich.close();
            std::cout << "cerrando fichero" << std::endl;
        }
    }
private:
    std::ofstream fich;
};

main()
{
    fichero fich("prueba.txt", "En función main\n");
}
```

Clase fichero:

- Constructor: abre fichero en el constructor para salida y agregar y escribe texto indicado.
- **explicit** indica que se requiere utilizar este constructor y no otro por defecto.
- En el destructor , cierra el fichero (si está abierto),

Se crea en el *stack* variable local de tipo fichero (utilizando el constructor establecido). Cuando termina la función main, el destructor de **fich** es llamado y el fichero se cierra “automáticamente”

Del C al C++

```
void log(std::string fich_nombre, std::string texto)
{
    fichero f(fich_nombre, texto);

    throw std::runtime_error("Error no especificado");
}

main()
{
    std::string nombre="prueba.txt";

    try {log(nombre, "Esta es una prueba");}

    catch (std::exception &ex)
    {
        std::cerr << ex.what() << std::endl;
    }
}
```

El esquema RAII funciona incluso en presencia de excepciones.

Se llama a la función susceptible de lanzar error en **try** y se captura error en **catch**.
El fichero abierto dentro de función **log** será cerrado convenientemente.



Del C al C++

- ▶ El patrón **RAII** viene implementado de múltiples formas en la biblioteca estándar:
 - ▶ Cada vez que utilizamos una clase contenedora (e.g. vector), los elementos del vector son reservados en el *heap* utilizando **new**. Cuando el vector sale del alcance del bloque donde fue definido, su constructor llama **delete**.
 - ▶ Debemos intentar no utilizar **new** y **delete** en nuestro código.
 - ▶ En la mayoría de las ocasiones, se puede evitar haciendo uso de las mencionadas clases contenedoras.
 - ▶ Cuando se considere necesario usarlos, se debe usar **unique_ptr** y **shared_ptr** de la librería estándar. El primero se utiliza cuando el objeto apuntado tiene un solo “dueño” el recurso se borra cuando el objeto es destruido. El segundo implementa un “**contador de referencia**” según a cantidad de “dueños” que tenga, cuando llega a cero el contador (el último “dueño” es destruido) se borra (**delete**) el recurso apuntado.



```

#include <iostream>
#include <memory>

struct sensor
{
    std::string referencia;
    double valor;
    ~sensor() {std::cout << "destruyendo\n";}
};

std::unique_ptr<sensor> llena_sensor(std::string referencia, double val)
{
    std::unique_ptr<sensor> p1(new sensor);

    p1->referencia=referencia;
    p1->valor=val;

    return p1; //es retornado por movimiento (único dueño)
}

main()
{
    std::unique_ptr<sensor> p2(llena_sensor("TT01",2.3));

    std::cout << p2->referencia << std::endl;

    std::unique_ptr<sensor> p3(std::move(p2));

    if (!p2)
        std::cout << "p2 ahora es nullptr\n";

    std::cout << p3->referencia << std::endl;
}

```

Crea **struct sensor** y lo apunta por puntero único, rellena campos y devuelve el puntero. El elemento no es destruido sino que es “movido” en el **return**

P2 se le asigna (por movimiento) lo que devuelve **llena_sensor**.

P3 se le asigna (por movimiento) lo que tenía p2. Al terminar **main** se destruye la única estructura que había sido creada en el *heap*


```

#include <iostream>
#include <memory>

struct sensor
{
    std::string referencia;
    double valor;
    ~sensor() {std::cout << "destruyendo\n";}
};

std::unique_ptr<sensor> llena_sensor(std::string referencia, double val)
{
    std::unique_ptr<sensor> p1(new sensor);

    p1->referencia=referencia;
    p1->valor=val;

    return p1; //es retornado por movimiento (único dueño)
}

main()
{
    std::unique_ptr<sensor> p2(llena_sensor("TT01",2.3));

    std::cout << p2->referencia << std::endl;

    std::unique_ptr<sensor> p3(std::move(p2));

    if (!p2)
        std::cout << "p2 ahora es nullptr\n";

    std::cout << p3->referencia << std::endl;
}

```

Salida por consola
del programa

```

TT01
p2 ahora es nullptr
TT01
destruyendo

```

Del C al C++

```
#include <iostream>
#include <memory>

struct sensor
{
    std::string referencia;
    double valor;
    ~sensor() {std::cout << "destruyendo\n";}
};

main()
{
    std::shared_ptr<sensor> p1(new sensor);
    p1->referencia = "TT01";
    std::shared_ptr<sensor> p2=p1;
    p2->valor =1;

    std::cout << "ref:" << p1->referencia << "\nval: " << p2-> valor<< std::endl;
}
```

Se crea p1 y se accede mediante puntero compartido

Se crea p2 y se le asigna p2. Ahora p1 y p2 apuntan al mismo objeto. El contador interno del puntero se incrementa.

Cuando p1 y luego p2 abandonan el alcance de **main**, contador llega a cero y se libera memoria objeto

```
ref:TT01
val: 1
destruyendo
```

Salida consola

Del C al C++

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>

class Recurso
{
public:
    bool salir=false;

    void mult(float factor)
    {
        std::lock_guard<std::mutex> guard(m);
        for (auto &elem: vec )
            elem*=factor;
    }

    void imprime()
    {
        std::lock_guard<std::mutex> guard(m);
        for (auto elem: vec )
            std::cout << elem << " ";
        std::cout << std::endl;
    }

private:
    std::mutex m;
    std::vector<float> vec {1,5,7,8,9,10,11,45,-12};
};
```

Clase Recurso a ser compartida entre varios *threads*

En lugar de utilizar métodos **lock** y **unlock** del **mutex**, se utiliza objeto auxiliar **lock_guard** que implementa patrón RAII

Incluye un vector y un **mutex** m que coordinará el acceso al vector de los métodos **mult()** e **imprime()**

Del C al C++

```
void ImprimeVec (Recurso &r)
{
    while (!r.salir)
    {
        std::this_thread::sleep_for(std::chrono::seconds(5));
        r.imprime();
    }
}

int main()
{
    Recurso rec;
    std::thread t(ImprimeVec, std::ref(rec));
    do
    {
        float f;

        std::cout << "Entre factor (negativo salir)" << std::endl;
        std::cin >> f;
        if (f >= 0)
            rec.mult(f);
        else
            rec.salir=true;
    }
    while (!rec.salir);
    t.join();
    return 0;
}
```

Función ha ser ejecutada en otro hilo. Recibe objeto tipo recurso por referencia.

Se crea hilo, se le indica la función a ejecutar y se le pasa el parámetro local a **main** (rec) por referencia.

Al salir, ejecutar **join** en el hilo.

main() se ejecuta concurrentemente y multiplica el vector del recurso **rec** por el número que de el usuario.