
UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

CS1103
Programación Orientada a Objetos 2
Unidad 1
Tema: Memoria Dinámica

Rubén Rivas

Conceptos Fundamentales de Programación

1. Punteros
2. Smart pointers
3. Conclusiones

Punteros

Un puntero en C/C++ se representa de la siguiente forma:

```
<Tipo de Dato>* <nombre del Puntero>;
```

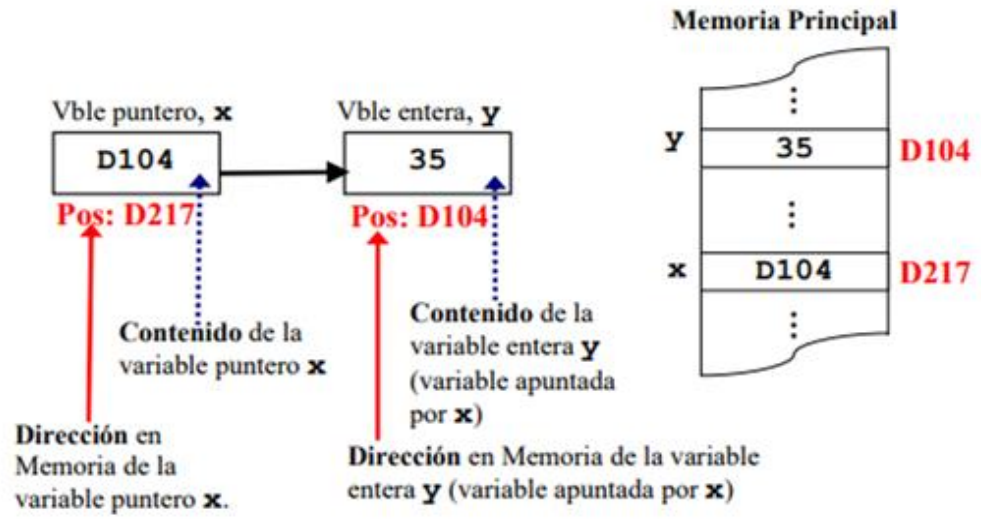
donde:

- **<Tipo de Dato>:** Representa uno de los tipo de dato válido en C/C++
- **<Nombre del Puntero>:** Expresión alfanumérica que sigue las mismas reglas de las variables en C/C++.

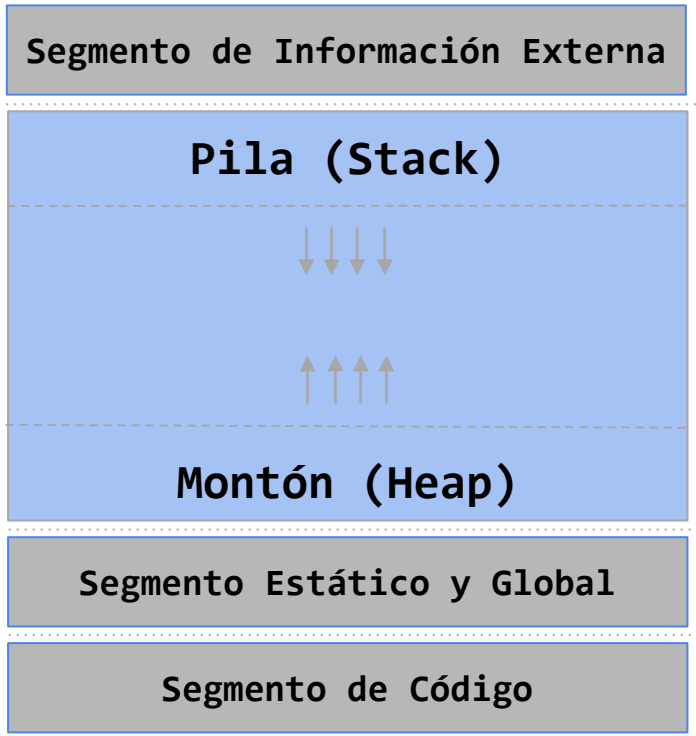
Ejemplo:

```
int* ptrEntero;
char* ptrCaracter;
```

```
int y = 35;
int* x = &y;
```



Gestión de memoria - Stack / Heap



Argumento externos (argc, argv)

Memoria automática donde se asigna las variables estáticamente.

Memoria donde se asigna variables dinámicamente.

Memoria donde se asigna variables globales y tipo static.

Instrucciones del Programa.

Gestión de memoria - Asignación/Libración en C++

- **Existen un operador que permiten asignar memoria dinámica:**
  - **new,** operador de uso genérico para asignación de memoria.

```
// ptr = new type; throw bad_alloc exception
double* ptr = new double;

// ptr = new (std::nothrow) type;
double* ptr = new (std::nothrow) double;
```

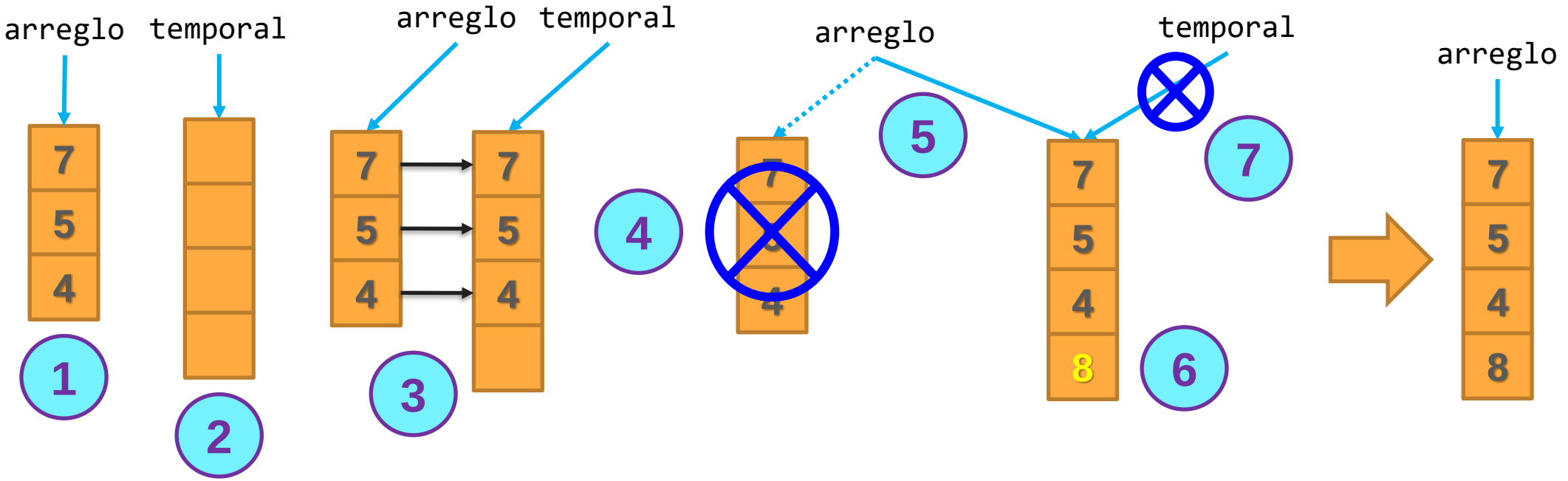
- **Existe sólo un operador (con sobrecarga) para la liberación:**
  - **delete,** para una colección de datos utiliza el modificador []

```
delete ptr;
// in case of ptr = new type[size]
delete [] ptr;
```

Característica sobre el uso del new y delete

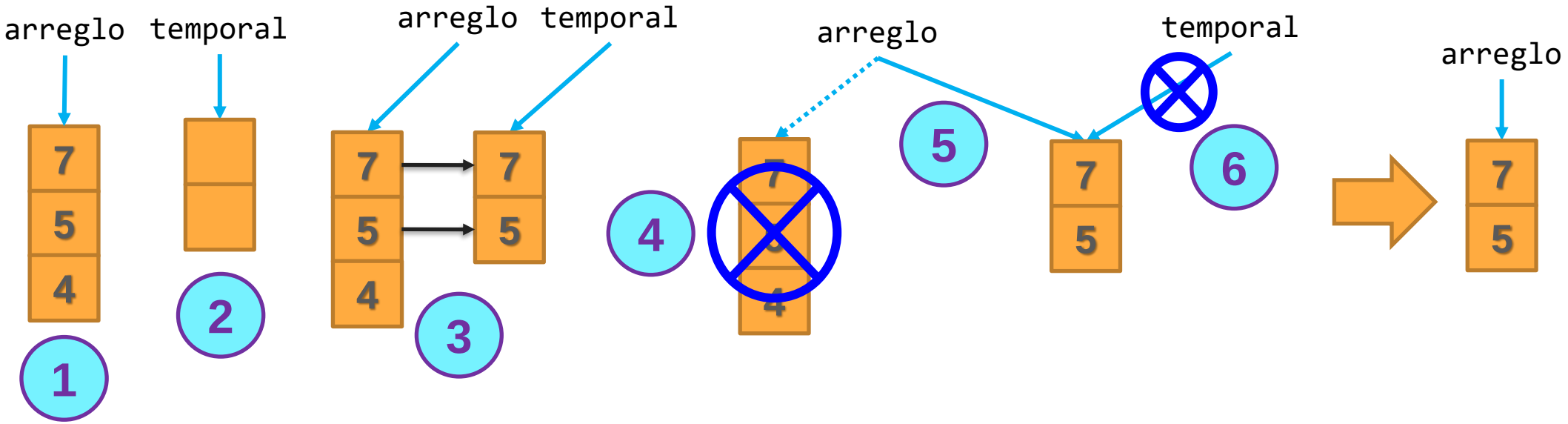
Aunque la asignación de memoria se puede realizar en cualquier instante y la referencia puede ser intercambiada entre punteros, la asignación de por si es fija, es decir que el espacio asignado no puede ser redimensionado. Esto involucra que si se requiere realizar acciones de redimensionamiento estas deben de ser simuladas.

Proceso para redimensionar – Agregar valor al final




```
1. int* arreglo = new int[3] {7, 5, 4};
2. int* temporal = new int[4]{};
3. for (int i = 0; i < 3; ++i)
        temporal[i] = arreglo[i];
4. delete [] arreglo;
5. arreglo = temporal;
6. arreglo[3] = 8;
7. temporal = nullptr;
```

Proceso para redimensionar – Eliminar valor al final




```
1. int* arreglo = new int[3] {7, 5, 4};
2. int* temporal = new int[2]{};
3. for (int i = 0; i < 2; ++i)
        temporal[i] = arreglo[i];
4. delete [] arreglo;
5. arreglo = temporal;
6. temporal = nullptr;
```

Reserved – al adicional

O(n)

```
void push_back(int value)
{
    int* temporal = new int[size_+1];
    for (int i = 0; i < size_; ++i)
        temporal[i] = data_[i];
    delete [] data_;
    data_ = temporal;

    data_[size_] = value;
    ++size_;
}
```

O(1) amortizado

```
void push_back(int value)
{
    if (size_ + 1 > reserved_) {
        reserved_ *= 2;
        int* temporal = new int[reserved_];
        for (int i = 0; i < size_; ++i)
            temporal[i] = data_[i];
        delete [] data_;
        data_ = temporal;
    }

    data_[size_] = value;
    ++size_;
}
```

Reserved – al borrar

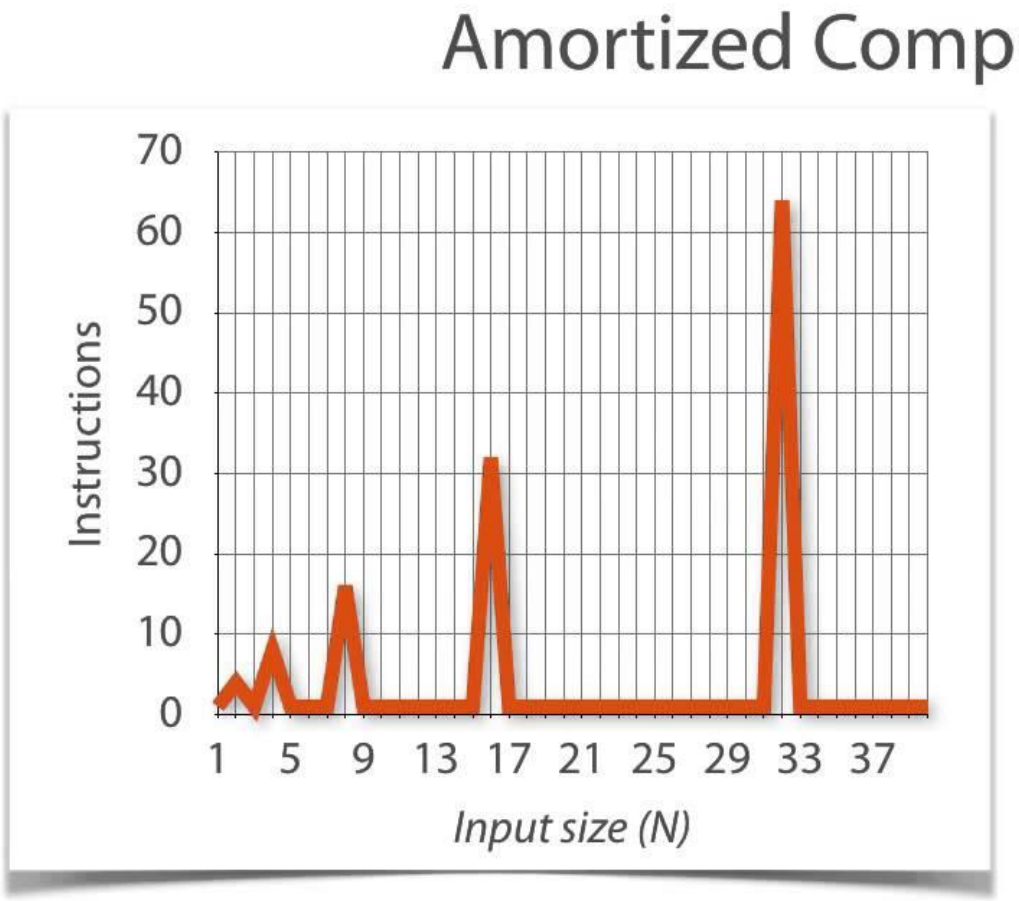
O(n)

```
void pop_back()
{
    int* temporal = new int[size_-1];
    for (int i = 0; i < size_-1; ++i)
        temporal[i] = data_[i];
    delete [] data_;
    data_ = temporal;

    --size_;
}
```

O(1) amortizado

```
void pop_back()
{
    if (size_ + 1 < reserved_ / 2) {
        reserved_ /= 2;
        int* temporal = new int[reserved_];
        for (int i = 0; i < size_-1; ++i)
            temporal[i] = data_[i];
        delete [] data_;
        data_ = temporal;
    }

    --size_;
}
```

Complejidad Amortizada



Complexity of multiple calls

Typically: *O*(1) per call

Housekeeping: *O*(*N*)

100

pluralsight

¿Por qué utilizar punteros?

- Permite acceder a espacios de memoria reservados en el **heap** o memoria dinámica.
- Se pueden utilizar como referencias a espacios fuera de la zona de memoria asignada a una función o método.
- Permite darle flexibilidad a la creación de nuevos tipos de datos.

Smart pointers

- Son **tipos abstractos de datos** que simulan el comportamiento de un puntero, de modo que realice un manejo automático de la memoria.
- Tienen como intención reducir las **fugas de memoria** (***memory leaks***), errores originadas por el olvido en la liberación de la memoria o por excepciones que no permitan eventual, durante el proceso de ejecución, liberar la memoria.
- Busca ser un método que tenga el menor costo computacional posible, tratando cumplir el principio de **zero-overhead** (No pagues por lo que no usas).
- Así también elimina la posibilidad de los **punteros colgados**. Aquellos punteros que almacenan direcciones incorrectas.

Técnicas usadas en su implementación

Los punteros inteligentes utilizan las siguientes características principales de una clase en C++:

- **Constructores y destructores**
- **Uso de memoria dinámica**
- **Uso de templates de clases**
- **Sobrecarga de operadores**
- **Encapsulamiento**

Tipos de punteros inteligentes

C++ clasifica los punteros inteligentes en 2 categorías y 3 tipos :

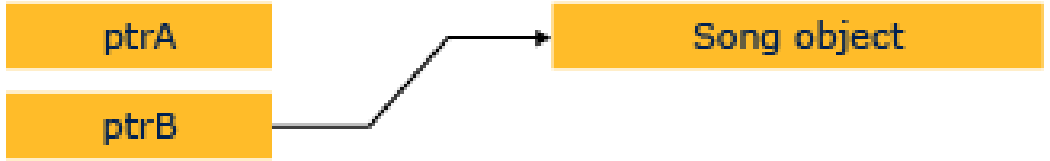
- Punteros no compartidos (**unique_ptr**)
- Punteros compartidos (**shared_ptr**) y
- Punteros compartidos débiles (**weak_ptr**)

Punteros no compartidos std::unique_ptr

- Reduce la complejidad de compartir la memoria entre varios punteros
- Esta diseñado para asignar a la memoria un SOLO puntero.
- Se encuentra definido en la cabecera **<memory>**
- Es bastante ligero solo requiere de una envoltura ligera.

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

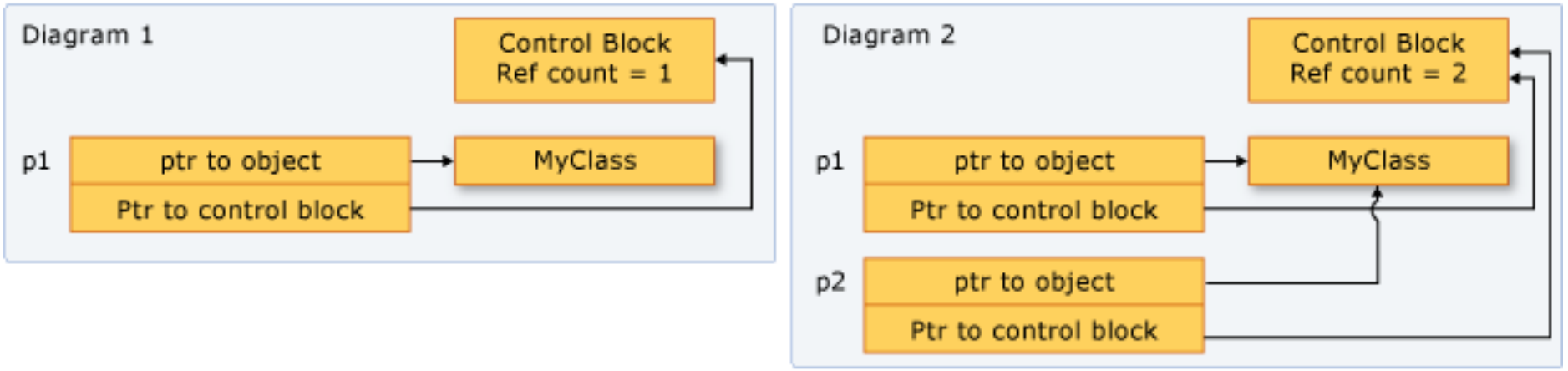

```
auto ptrB = std::move(ptrA);
```



Fuente: *https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019*

Punteros compartidos (std::shared_ptr)

- Están diseñados para escenarios complejos donde se requiere compartir **la propiedad de la memoria.**
- Cuenta adicionalmente con una **referencia a un contador** que registra el número de propietarios.
- Cuando el valor de contador alcanza el **valor de cero**, se libera la memoria.



Fuente: *https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019*

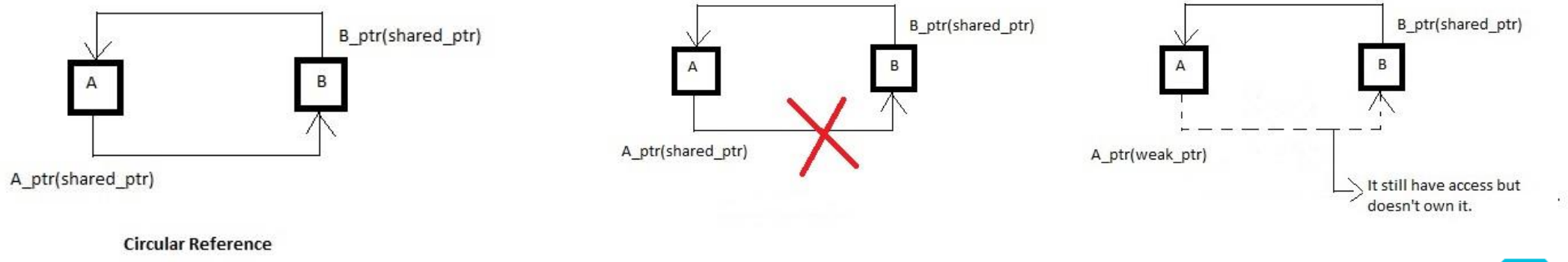
Métodos de punteros compartidos (std::shared_ptr)

- **`reset()`,** libera la referencia al objeto manejado, equivalente a asignar a un puntero tradicional el valor **`nullptr`**.
- **`used_count()`,** retorna el número de **`shared_ptr`** que comparten la propiedad de un objeto manejado.
- **`get()`,** retorna el puntero original.
- **`operator bool()`,** retorna si esta vacío, equivalente a la verificar si un puntero tradicional apunta a **`nullptr`**.

Fuente: *https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019*

Punteros compartidos débiles (std::weak_ptr)

- Similar a los punteros **shared_ptr**, puede acceder a ellos sin causar el incremento del contador de referencias, actúa como un observador.
- Es útil en casos en que existe referencia cíclica entre instancias de objetos tipo **shared_ptr**.



Fuente: *https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/*

Métodos de punteros compartidos (std::weak_ptr)

- **`lock()`,** crea un puntero del tipo **shared_ptr** del objeto que maneja, si no tuviera asignado algún **objeto manejado** entonces retorna vacío.
- **`expired()`,** verifica si la referencia al **objeto manejo** fue borrada y es equivalente a **used_count() == 0**.

Fuente: *https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019*

Punteros compartidos débiles (weak_ptr)

```
// weak_ptr::lock example
#include <iostream>
#include <memory>

int main () {
  std::shared_ptr<int> sp1,sp2;
  std::weak_ptr<int> wp;
                                        // sharing group:
                                        // --------------
  sp1 = std::make_shared<int> (20);     // sp1
  wp = sp1;                             // sp1, wp

  sp2 = wp.lock();                      // sp1, wp, sp2
  sp1.reset();                          //      wp, sp2

  sp1 = wp.lock();                      // sp1, wp, sp2

  std::cout << "*sp1: " << *sp1 << '\n';
  std::cout << "*sp2: " << *sp2 << '\n';

  return 0;
}
```

Conclusiones

- Los punteros brindan mayor flexibilidad en el uso de la memoria
- Un uso incorrecto podría generar problemas (**memory leaks**, **dangle pointers**)
- C++ brinda una alternativa a los punteros convencionales, conocida como punteros inteligentes
- Los punteros inteligentes se clasifican en 2 categorías, punteros no compartidos y punteros compartidos.