

Smart Pointers



TPV2
Samir Genaim

UNIVERSIDAD
COMPLUTENSE
MADRID
Lecturas
Samir Genaim
TPV2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

Memoria Dinámica

- ◆ Normalmente queremos liberar la memoria cuando no se va a usar más durante la ejecución para permitir que otros la usen ...
- ◆ a veces el manejo de la **memoria dinámica** (quien va a libera la memoria y cuando) **puede ser una tarea muy complicada.**
- ◆ En lenguajes como C (y en principio C++) el programador es el único responsable del manejo de la memoria dinámica ...
- ◆ En languages como Java es mucho más fácil porque el programador solo **crea memoria** y el **colector de basura** la libera cuando no hay ningún referencia a esa memoria ...
- ◆ Usar un **colector de basura** normalmente afecta a las prestaciones, pero muchas veces es un precio que estamos dispuestos a pagar para liberarnos de esa tarea ...

Smart Pointer

- ◆ Un **smart pointer** es un tipo abstracto de datos que simula el comportamiento de un **puntero normal** (raw pointer) pero tiene características adicionales como **recolector de basura** automático ...
- ◆ Normalmente es parte de las librerías en lenguajes que no tienen colector de basura ...
- ◆ C++ tiene varios tipos de smart pointers, vamos a ver dos:
 - **Unique pointer**: simula un raw pointer único, es decir no se puede tener dos referencias a esa memoria (no se puede copiar, **no tiene constructora de copia**). Libera la memoria al salir del **scope** correspondiente ...
 - **Shared pointer**: simula un raw pointer compartido y mantiene un contador de referencias a esa memoria, se puede pasar a cualquiera y se destruye automáticamente cuando ninguno tiene referencia a esa memoria ...

Unique Pointers

```
void f() {  
    A *x = new A(...);  
    x->foo();  
    ...  
    delete x;  
}
```

Se borra el puntero al objeto A cuando se borra la instancia del unique_ptr - al salir de método

```
void f() {  
    std::unique_ptr<A> x(new A(...));  
    x->foo(); // use x like if it were A*  
    ...  
    // will be deleted automatically  
    // when leaving the scope  
}
```

```
void f() {  
    auto x = std::make_unique<A>(...);  
    x->foo(); // use x like if it were A*  
    ...  
    // will be deleted automatically  
    // when leaving the scope  
}
```

Unique Pointers

```
void f() {  
    if (...) {  
        A *x = new A(...);  
        x->foo();  
        ...  
        delete x;  
    }  
    ...  
}
```

Se borra el puntero al objeto A cuando se borra la instancia del unique_ptr - al salir de la parte “then” del “if”

```
void f() {  
    if (...) {  
        std::unique_ptr<A> x(new A(...));  
        x->foo();  
        ...  
    }  
    ...  
}
```

```
void f() {  
    if (...) {  
        auto x = std::make_unique<A>(...);  
        x->foo();  
        ...  
    }  
    ...  
}
```

// will be deleted automatically
// when leaving the scope

Unique Pointers

Muy útil cuando el flujo de control es complicado, es decir se puede salir del método en varios puntos, o incluso salir con una excepción, y queremos garantizar que el puntero se borra en todos estos casos

```
void f() {  
    auto x = std::make_unique<A>(...);  
    if (...) {  
        throw ...  
    } else {  
        y->foo(); // might throw exception  
        ...  
        return ...  
    }  
    ...  
    return ...  
}
```

Unique Pointers - caso de uso II

Es útil para tener containers (vector, list, etc.) de punteros que se borran cuando se eliminan del container

```
std::vector<std::unique_ptr<A>> v;  
  
v.emplace_back(new A());  
  
auto x = std::make_unique<A>();  
v.emplace_back(std::move(x));  
  
v.erase(std::remove_if(v.begin(),  
                      v.end(),  
                      [](<const std::unique_ptr<A> &o) { ... }),  
        v.end());
```

Unique Pointers

- ◆ Unique pointers no se pueden pasar por copia ni asignar, porque queremos garantizar que al salir del scope nadie tiene referencia al puntero correspondiente porque lo vamos a borrar ...
- ◆ Unique pointers se pueden mover de un objeto a otro ...
- ◆ Se puede resetear el punter gestionado usando **reset()**, borra el puntero actual y empieza a gestionar o
- ◆ Se puede consultar el puntero gestionado usando **get()**.
- ◆ El uso correcto es **vuestra responsabilidad**, porque usando **get()** siempre podemos consultar el puntero y pasarlo a cualquiera (eso normalmente no se debe hacer)

Unique Pointer - Implementación

```
template<typename T>
class uptr {
    T *p_;
public:
    uptr(const T&) = delete;
    uptr<T>& operator=(const uptr<T>&) = delete;

    uptr() : p_(nullptr) { }
    uptr(T *p) : p_(p) { }
    uptr(uptr<T> &&o) {
        p_ = o.p_;
        o.p_ = nullptr;
    }
    ...
};
```

El puntero gestionado

No se puede copiar

Por defecto no gestiona nada

gestiona un puntero p a *T

Move constructor, 'o' ya no gestiona nada

Unique Pointer - Implementación

```
template<typename T>
class uptr {
    ...
    uptr<T>& operator=(uptr<T> &&o) {
        reset(nullptr);
        p_ = o.p_;
        o.p_ = nullptr;
    }
}
```

Move assignment, 'o' ya no gestiona nada

```
virtual ~uptr() { reset(nullptr); }
```

Destructor, se borra el puntero actual llamando a reset

```
void reset(T *p) {
    if (p_ != nullptr)
        delete p_;
    p_ = p;
};
```

Borra el puntero actual y empieza a gestionar p

Unique Pointer - Implementación

```
template<typename T>
class uptr {
...
T* operator->() {
    return p_;
}
T& operator*() {
    return *p_;
}
T* get() {
    return p_;
};
```

Para poder usar el unique pointer
como se fuera un puntero de tipo T

Consultar el puntero gestionado

Shared Pointers

Cuando creamos un puntero y lo pasamos a muchas partes del programa, a veces es difícil decidir quién tiene que borrarlo. Usando shared pointers se borra automáticamente cuando todos dejan de tener la referencia.

```
void f() {  
    A *x = new A(...);  
  
    ...  
    g(x);  
    ...  
    y.m(x);  
    ...  
}
```

```
void f() {  
    std::shared_ptr<A> x(new A(...));
```

```
    ...  
    g(x);  
    ...  
    y.m(x);  
    ...  
}
```

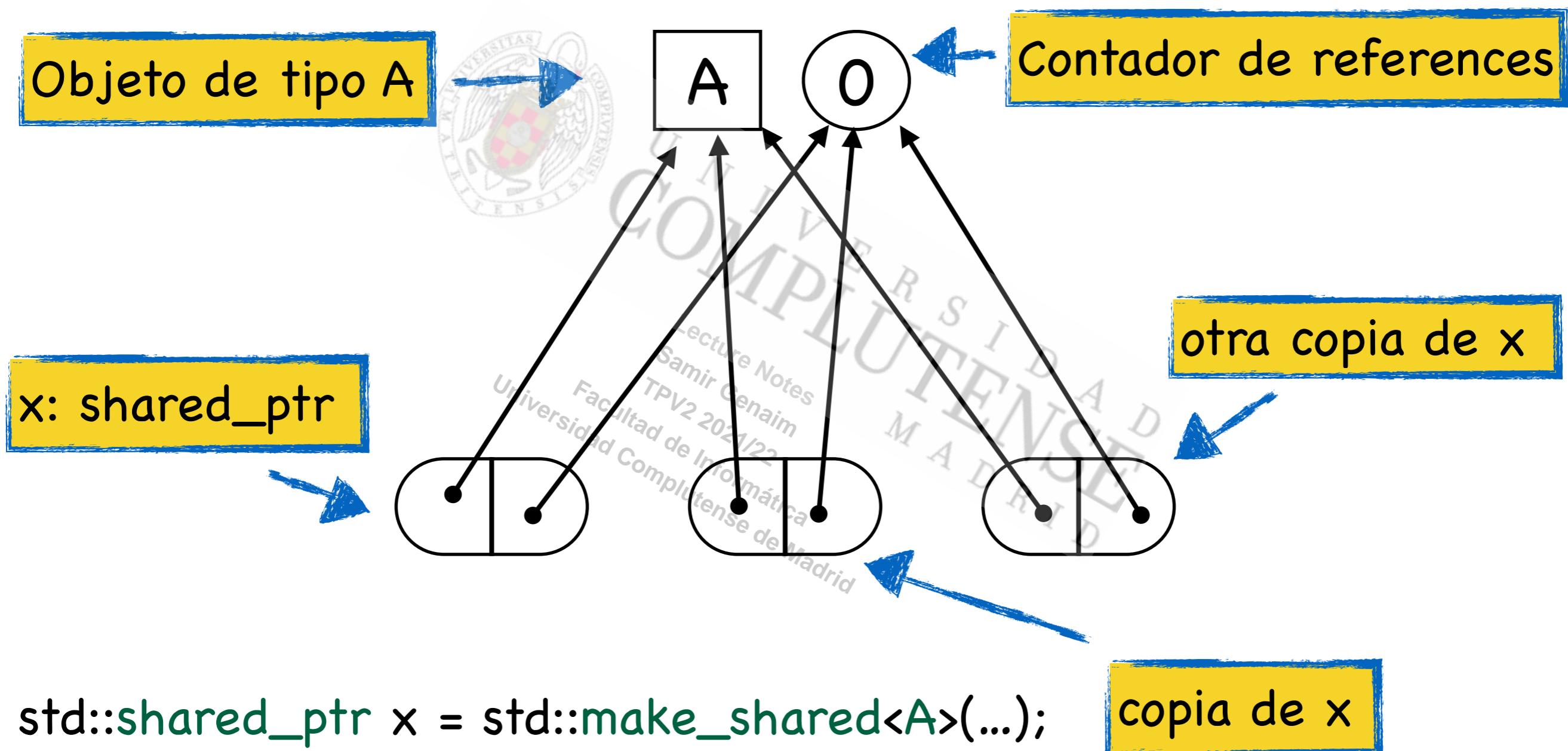
```
void f() {  
    auto x = std::make_shared<A>(...);
```

```
    ...  
    g(x);  
    ...  
    y.m(x);  
    ...  
}
```

Salvo autorización expresa, los materiales
entregados a estudiantes por cualquier medio
durante la carrera sólo se podrán utilizar
para el estudio de la asignatura correspondiente
en la Universidad Complutense de Madrid.
La publicación o distribución posterior
(incluida la divulgación en redes sociales
o servicios de comunicación de protección
de datos y/o de propiedad intelectual)
puede vulnerar la normativa de protección
de datos y/o de propiedad intelectual
que no tenga la extensión ucm.es,
avísanos en denunciacontenidos@ucm.es
o reportcontent@ucm.es
y encontrarás esta material en otro sitio web
genera responsabilidad de la persona infractora.

Shared Pointers

La idea es envolver el puntero con un objeto de tipo `shared_ptr` que lo gestiona. La clase `shared_ptr` mantiene un contador de copias que a esa referencia, y cuando el contador llega a ser 0 se borra el puntero.



Shared Pointer - Implementación

```
template<typename T>
class shptr {
    T *p_;
    int *c_;
public:
    shptr() : p_(nullptr), c_(nullptr) {}
    shptr(T *p) : p_(nullptr), c_(nullptr) { reset(p); }

    shptr(const shptr<T> &o) {
        p_ = o.p_;
        c_ = o.c_;
        if (p_ != nullptr) (*c_)++;
    }
    ...
}
```

El puntero gestionado y su contador de referencias

Por defecto no gestiona nada

Gestionar un puntero p llamando a reset(p) que actualiza p_ y c_

Copy constructor, incrementamos el contado de referencias si ya está gestionando un puntero

Shared Pointer - Implementación

```
template<typename T>
class shptr {
    ...
    shptr(shptr<T> &&o) {
        p_ = o.p_;
        c_ = o.c_;
        o.p_ = nullptr;
        o.c_ = nullptr;
    }

    ~shptr() {
        reset(nullptr);
    }

    ...
}
```

Move constructor, no se incrementa el contado de referencias porque añadimos una y quitamos una

Destructor, deja de manejar el puntero actual llamando a reset (que se encarga de quitar 1 del contador de referencias, etc.)

Shared Pointer - Implementación

```
template<typename T>
class shptr {
    ...
    shptr<T>& operator=(const shptr<T> &o) {
        reset(nullptr);
        p_ = o.p_;
        c_ = o.c_;
        if (p_ != nullptr)
            (*c_)++;
        return *this;
    }
    ...
}
```

Copy assignment, es como el copy constructor pero deja de gestionar el puntero actual llamando a `reset(nullptr)`.

Shared Pointer - Implementación

```
template<typename T>
class shptr {
    ...
    shptr<T>& operator=(shptr<T> &&o) {
        reset(nullptr);
        p_ = o.p_;
        c_ = o.c_;
        o.p_ = nullptr;
        o.c_ = nullptr;
        return *this;
    }
    ...
}
```

Move assignment, es como el move constructor pero deja de gestionar el puntero actual llamando a `reset(nullptr)`.

Shared Pointer - Implementación

```
template<typename T>
class shptr {
...
void reset(T *p) {
    if (p_ != nullptr) {
        (*c_)--;
        if (*c_ == 0) {
            delete p_;
            delete c_;
        }
    }
    p_ = p;
    if (p != nullptr) c_ = new int(1);
    else c_ = nullptr;
}
...
}
```

Quita 1 del contador de referencias, y si llega a ser 0 borramos el puntero y el contador de referencias

Empezar a gestionar p

Shared Pointer - Implementación

```
template<typename T>
class shptr {
    ...
    T* operator->() {
        return p_;
    }
    T& operator*() {
        return *p_;
    }
    T* get() {
        return p_;
    }
}
```

Para poder usar el shared pointer como se fuera un puntero de tipo T

Consultar el puntero gestionado

Shared Pointers

- ◆ Shared pointers se pueden pasar por copia y asignar ...
- ◆ Shared pointers se pueden mover de un objeto a otro ...
- ◆ Se puede resetear el puntero gestionado usando **reset()**
- ◆ Se puede consultar el puntero gestionado usando **get()**
- ◆ El uso correcto es **vuestra responsabilidad**, porque usando **get()** siempre podemos consultar el puntero y pasarlo (eso normalmente no se debe hacer)

Custom Deleters

A veces nos interesa borrar el puntero de otra manera o simplemente hacer algo más de borrarlo (por ejemplo cuando usamos gestión de memoria propio, los punteros no se borran sino se devuelven al gestor de memoria)

```
std::unique_ptr<A, std::function<void(A*)>>  
    x( new A(...), [](A* p) {  
        cout << "deleting " << p << endl;  
        delete p;  
    });
```

El smart pointer llama al λ-expr. En lugar de liberar la memoria

```
std::shared_ptr<A, std::function<void(A*)>>  
    x( objectPool.create(), [](A* p) {  
        objectPool.free(p);  
    });
```

Ejercicio: modificar las clases uptr y shptr para poder usar custom deleters

Smart Pointers y Arrays

Arrays al final son punteros, pero cuando liberamos la memoria correspondiente hay que usar `delete[]`. A partir de C++17 smart pointers pueden ser de arrays.

```
std::shared_ptr<int[]> x(new int[10]);
```

```
x[0] = 1;
```

```
x[1] = x[0] + 5;
```

```
std::unique_ptr<int[]> x(new int[10]);
```

```
x[0] = 1;
```

```
x[1] = x[0] + 5;
```

Antes de C++17, se podía simular con punteros y usar custom deleters para borrarlos

```
std::unique_ptr<int, std::function<void(int*)>>
    x(new int[10], [](int *p) { delete[] p; });
```