

# Resumen del libro Effective C++

feserr

1 de agosto de 2018

# Item 1: View C++ as a federation of languages

## Item 2: Prefer consts, enums, and inlines to #defines. I

- ▶ Los **defines** pueden no reflejarse jamás en el compilador, este puede generar múltiples instancias de él y que no lo encapsule.
- ▶ Los **define** no se pueden encapsular.
- ▶ Sobre **const**:
  - ▶ Solo se hace una instancia de la variable.
  - ▶ Si se quiere que sea única en la clase, hay que declararla **static**. Para ello será necesario definirlo, por ejemplo:

```
// .h
...
private:
    static const int num;
    ...

// .cpp
const int clase::num;
```

## Item 2: Prefer consts, enums, and inlines to #defines.

### ► Respecto a los **enum**

- Se hace uso del **enum hack** para definir variables, por ejemplo:

```
// .h
enum {total = 5;}
int nums[total];
```

- Su uso es parecido al de **define**, pero con estos se puede obtener la dirección de memoria.

### ► En cuanto a los **inline**

- Se usan para sustituir la creación de macros con **define**.
- El uso de **define** hace que no se controlen bien ciertas ejecuciones, dado que si se incrementa una variable de la macro, esta puede ocurrir en distintos casos. Por ejemplo:

```
#define max(a,b) f((a) > (b) = (a) : (b))
max(++a, b);      // "a" se incrementa dos veces
max(++a, b+10);   // "a" se incrementa solo una vez
```

- Con **inline** se permite la creación de macros privadas.

## Item 2: Prefer consts, enums, and inlines to #defines. III

- ▶ Para constantes simples, usar objetos **const** o **enums** en vez de **#defines**.
- ▶ Para funciones tipo macros, usar **inline** en vez de #defines.

## Item 3: Use const whenever possible I

- ▶ Se verifican en tiempo de compilación.
- ▶ Afectan al tipo de su izquierda.

```
const char *p = "Hello"; // const data  
char * const p = "Hello"; // const pointer
```

- ▶ **const\_iterator** es como **const T**. Se puede usar para que el valor que se devuelva no pueda ser modificado, por ejemplo en una operación.
- ▶ Se puede usar cuando se quiere devolver una variable constante o normal. Por lo que se crea la función que devuelve el valor constante y luego para devolverla en su forma normal, haremos que el método **non-const** sea el que llame al constante de la siguiente manera.

## Item 3: Use const whenever possible II

```
class TextBlock {
public:
    ...
    const char& operator[](std::size_t position) const {
        ...
        return text[position];
    }

    char& operator[](std::size_t position) {
        return
            const_cast<char&>(                                // quitamos const en
                                                                // el tipo de retorno
            static_cast<const TextBlock&>(*this) // ponemos const al
            [position]                               // tipo de *this
                                                    // llamamos a la
                                                    // version
                                                    // const de op[]
            );
    }
    ...
};
```

## Item 3: Use const whenever possible III

- ▶ Si se pone antes de las llaves en una función, entonces especificamos que esa definición será constante y no se puede modificar nada en la declaración (**bitwise constness**), por lo que para modificar una variable en la misma, habremos de señalar que es **mutable** (**logical constness**).
- ▶ Usar **const** ayuda al compilador a detectar errores de uso. **const** se puede aplicar a cualquier tipo de objetos, parámetros de las funciones, tipos de retorno y al conjuntos de miembros de una función.
- ▶ Los compiladores fuerzan a aplicar **bitwise constness**, pero se debe programar usando **logical constness**.
- ▶ Cuando los miembros de una función son **const** y **non-const** tienen idéntica implementación, la duplicación de código puede ser evitada haciendo que la versión **non-const** llame a la versión **const**.



## Item 4: Make sure that objects are initialized before they are used

- Hay que inicializar siempre las variables en el **constructor**, tanto en los que aceptan parámetros como en los de por defecto.

## Item 5: Know what functions C++ silently writes and calls

- ▶ Cuando creas una clase, sino declaras el **constructor**, **destructor**, **copy constructor** y **copy assignment operator** el compilador los creara **públicos** por ti.
- ▶ Las funciones **copy constructor** y **copy assignment operator** básicamente copiaran los elemento **no estáticos** de un objeto al otro. Pero esto solo lo hará cuando sea posible, es decir, si se intenta copiar un **std::string** usará el **copy constructor** o **copy assignment operator** de esta clase, ya que esta declarado, o en el caso de que sea un elemento simple (p.e int) copiará los bits de uno a otro. En el caso de que sea una variable que no se pueda, p.e una variable **const**, el compilador no generara automáticamente estos métodos.
- ▶ Es posible que el compilador genere implícitamente el **constructor**, **copy constructor**, **copy assignment operator** y **destructor**.

## Item 6: Explicitly disallow the use of compiler generated functions you do not want

- ▶ Si no queremos que se hagan copias de nuestros objetos, debemos declarar los métodos **copy constructor** y **copy assignment operator** privados, de este modo el error saldrá en la fase de enlace.
- ▶ Si queremos hacer que el error sea en compilación, deberemos crear una clase como la siguiente.

```
class Uncopyable {
protected:
    Uncopyable() {}
    ~Uncopyable() {}
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator=(const Uncopyable&);
};
```

De este modo, el objeto que no queramos que sea copiado, haremos que herede de forma privada del objeto anterior.

## Item 7: Declare destructors virtual in polymorphic base classes I

- ▶ Si una clase base no declara virtual su destructor, cuando el objeto que herede de este e intente eliminarlo solo eliminara la memoria de la clase padre y no del objeto hijo.
- ▶ Si una clase no tiene ninguna función virtual no esta orientada a ser una clase base o padre. Por lo tanto, su destructor no tiene que ser virtual. Se deben obviar el uso de virtual ya que genera información extra, por ejemplo en la siguiente clase.

```
class Point {  
    public:  
        Point() {}  
        ~Point() {}  
    private:  
        int x, y;  
};
```

## Item 7: Declare destructors virtual in polymorphic base classes II

- ▶ Si un `int` ocupa 32 bit, esta clase cabrá en un registro de 64 bits. En el caso de implementar alguna función **virtual** se requiere que el objeto lleve información de que objeto se lanza la función **virtual**. Esta información toma la forma de un puntero llamado `vptr` ("virtual table pointer"). Este apunta a una array de punteros a funciones llamado `vtbl` ("virtual table"); cada clase con funciones virtuales tiene asociada una `vtbl`. En el caso de que nuestra clase `Point` tuviese una función virtual ocuparía 64 bits por los dos enteros mas la mitad de dicha suma, en total unos 96 bits. Se estaría perdiendo memoria.
- ▶ Hay que tener cuidado con heredar de clases de la **STL**, **string** o todos los tipos de contenedores. Estos no tienen declarado el **destructor** de forma **virtual**.

## Item 7: Declare destructors virtual in polymorphic base classes III

- ▶ Si quieres crear una clase base como abstracta pero no tienes ninguna función virtual pura, la solución es crear su **destructor** como **virtual puro**. De este modo sera abstracta y no nos tendremos que preocupar por problemas al destruirla. La única desventaja es que debemos definir el destructor en todos los hijos.
- ▶ Solo las clases bases que vayan a ser poliformicas se les puede aplicar la regla del **destructor virtual**. Aun así, si una clase tiene una función **virtual** su **destructor** ha de serlo también.

## Item 8: Prevent exceptions from leaving destructors

- ▶ En un **destructor** no pueden haber llamadas que generen excepciones. Dependiendo de que tipo sea, puede abortar el programa o dar un comportamiento indefinido y dejar memoria por liberar (corrupta).
- ▶ Se puede solucionar usando `try` y `catch` parando la ejecución cuando salta una excepción o continuar, dejando memoria por liberar, y avisar al programador.
- ▶ Una manera de solucionarlo es crear un método que libere la memoria y en el **destructor** comprobar que si no esta liberada usar lo expuesto en el punto anterior. No es una manera muy elegante de solucionarlo, ya que dejas la responsabilidad al programador, pero le das la oportunidad de que trate los errores a que no tengan oportunidad de reaccionar a los mismos.

## Item 9: Never call virtual functions during construction or destruction

- ▶ Si creamos una clase base que el **constructor** o **destructor** llaman a una función virtual para modificar unas variables, cuando un objeto herede de esta dará comportamientos inesperados. El problema radica en que cuando un objeto que hereda de otro se construye o se destruye primero llama a los métodos de la clase base y luego a los derivados, esto es porque antes de llamar a sus métodos derivados, el objeto es tratado como si fuese de la clase base, nada del objeto derivado existe. Por lo tanto, si llama a los métodos virtuales, por mucho que los redeclaremos en el derivado, el seguirá llamando a los de la clase base, y esto hará que las variables modificadas se queden como **undefined**.
- ▶ Esto también pasa si el constructor de la clase base llama a un método que a su vez este llama a un método virtual. Con esto conseguimos engañar al compilador para que no nos pueda avisar en la fase de compilación.



## Item 10: Have assignment operators return a reference to `*this`

- ▶ Las asignaciones en C++ pueden encadenarse.

`x = y = z = 15;`

También hay que decir que son *right-associative*, por lo tanto:

`(x = (y = (z = 15)))`;

En este ejemplo el resultado de `z` se devuelve y lo coge `y` que a su vez lo devuelve y lo obtiene `x`

- ▶ Para conseguir esto debemos devolver **`*this`** siempre que declaremos o sobrescribamos métodos de asignación.

```
class Point {  
public:  
    ...  
    Point& operator=(const Point& p)  
    {  
        ...  
        return *this;  
    }  
};
```

## Item 11: Handle assignment to self in operator= I

- Es probable que sin darnos cuentas hagamos un *self assignment*, por ejemplo:

```
// Ejemplo 1
class Widget { ... };
Widget w;
...
w = w; // asignación a si mismo
```

```
// Ejemplo 2
a[i] = a[j]; // asignación potencial a si mismo
```

```
// Ejemplo 3
class Base { ... };
class Derived: public Base { ... };
void doSomething(const Base& rb, // rb y *pd puedes ser
Derived* pd);                  // el mismo objeto
```

## Item 11: Handle assignment to self in operator= II

- ▶ Ejemplo 1: Vemos como claramente asignamos el mismo objeto a si mismo.
- ▶ Ejemplo 2: En el caso de que *i* y *j* apunten al mismo objeto se causara un *self assignment*.
- ▶ Ejemplo 3: En temas de herencia, podemos tener que una referencia o puntero del objeto base sea el mismo que el del objeto derivado.

## Item 11: Handle assignment to self in operator= III

- Esto pasa si nuestro método de asignación es del siguiente modo:

```
Widget&
Widget::operator=(const Widget& rhs) // unsafe impl. de
    operator=
{
    delete pb;                      // deja de usar el bitmap actual
    pb = new Bitmap(*rhs.pb); // empieza a usar una copia del
        bitmap
                                // de rhs
    return *this;                // ver Item 10
}
```

## Item 11: Handle assignment to self in operator= IV

- En el caso de que estemos usando esto con el mismo objeto, cuando eliminamos el `bitmap` actual también eliminamos el otro, dado que es el mismo. Una solución sería:

```
Widget&
Widget::operator=(const Widget& rhs) // unsafe impl. de
    operator=
{
    if (this == &rhs) return *this; // Si es el mismo no hace
        nada

    delete pb;
    pb = new Bitmap(*rhs.pb);

    return *this;
}
```

- El problema de esta solución es que es `exception-unsafe`. Puede que el nuevo `bitmap` contenga una excepción (porque no queda memoria o porque el constructor lance una), en este caso apuntará a un objeto eliminado (**dangling pointer**).

## Item 11: Handle assignment to self in operator= V

- ▶ Si queremos solucionar el problema de *self assignment* y `exception-unsafe` deberemos guardamos una copia del objeto original, cambiarlo por el nuevo objeto y eliminar la copia. También podríamos proceder construyendo un objeto temporal con el objeto nuevo y luego intercambiarlo con el original. Podemos ver la solución en el siguiente ejemplo.

## Item 11: Handle assignment to self in operator= VI

```
class Widget {
...
void swap(Widget& rhs); // intercambia los datos de *this y rhs
...
};

Widget& Widget::operator=(const Widget& rhs) {
    Widget temp(rhs);    // hace una copia del data de rhs
    swap(temp);          // intercambia los datos de *this con
                        // el de la copia

    return *this;
}

// ó

Widget& Widget::operator=(Widget rhs) { // rhs es una copia del
    swap(rhs);                          // objeto pasado
                                        // intercambia los
                                        // datos de
    return *this;                       // *this por el de la
    copia
}
```

## Item 11: Handle assignment to self in operator= VII

- El segundo ejemplo hace lo mismo que el primero pero construye la copia en el parámetro al ser pasado por valor. Este método puede ser mejor dado que los compiladores a veces generan código mejor optimizado.



## Item 12: Copy all parts of an object

- ▶ Cuando sobrecargamos la función de *copy assignment* tenemos que acordarnos de copiar todas las variables y objetos al nuevo.
- ▶ Es fácil olvidarnos de copiarlo todo cuando la clase es hija de otra. En este caso debemos llamar a la función de *copy assignment* de la clase padre, porque sino no estaremos copiando todos los elementos, por el hecho de que cuando nuestra clase es hija de otra, esta también es una clase padre por dentro.
- ▶ Al copiar los datos de las variables que son objetos y de la clase padre hay que hacerlo usando las funciones de copia apropiadas.
- ▶ Cuando el comportamiento del *copy constructor* y *copy assignment* es igual es mejor crear una tercera función que sea llamada por los dos.
- ▶ Hay que tener especial cuidado con no llamar al *copy constructor* en el *copy assignment* y viceversa si no queremos que la memoria se corrompa o surjan comportamientos extraños.

## Item 13: Use objects to manage resource

- ▶ Cuando creamos un objeto dinámico siempre ha de ser eliminado antes de que el programa se cierre si no queremos dejar memoria corrupta.
- ▶ En el caso de crear un objeto en una función y eliminarla al final puede crear muchos problemas. En un principio estamos eliminando el objeto que hemos creado, pero solo cuando ha llegado al final de la misma. Si se activase un retorno antes estaríamos dejando memoria por liberar. También puede ocurrir si creamos y eliminamos objetos dentro de un bucle y en este hay *break* y *goto*. No hay que olvidarse de funciones que pueden lanzar excepciones.
- ▶ Para ello existe *auto\_ptr* y *shared\_ptr*. Estos dos objetos de la STL se encargan de liberar la memoria cuando el programa a terminado, están fuera del alcance, y en los casos descritos anteriormente. Eso si, estos objetos llaman a *delete* para liberar su memoria, por lo que no son candidatos de ser usados en *arrays*.
- ▶ Algo positivo de *auto\_ptr* y *shared\_ptr* es que siguen la política RAll y que siempre serán eliminados no importa el comportamiento que tenga el programa.

## Item 14: Think carefully about copying behavior in resource-managing classes I

- ▶ No los recursos se generan dinámicamente, y para ellos usar *auto\_ptr* o *shared\_ptr* no sirven, por lo que tendrás que crear uno propio.
- ▶ Si, por ejemplo, queremos uno que manipule objetos del tipo *Mutex* que ofrecen la función *lock* y *unlock*:

```
class Lock {  
public:  
    explicit Lock(Mutex *pm) : mutexPtr(pm) {  
        lock(mutexPtr); } // adquirimos el recurso  
  
    ~Lock() { unlock(mutexPtr); } // liberamos el recurso  
private:  
    Mutex *mutexPtr;  
};
```

## Item 14: Think carefully about copying behavior in resource-managing classes II

- ▶ En el caso de usar este objeto, cuando se salga del alcance de la función, se desbloqueará automáticamente por el hecho de que su destructor hace eso.
- ▶ El problema que puede crear es si copiamos el objeto a otro nuevo, si uno se destruye desbloqueará el *mutex* y aun queda otro objeto que sigue bloqueado.
- ▶ Para ello hay diferentes soluciones:
  - ▶ Prohibir la copia. Declararemos privado las funciones de copia.
  - ▶ Llevaremos una cuenta de cuantos objetos se han creado y solo llamar al destructor cuando no quedan más.
- ▶ El comportamiento de la copia en las clases RAII más comunes es no permitir la copia del recursos en si. Para ello llevan la cuenta de el número de referencias que se han hecho, aun así existen otros tipos de comportamiento posibles.

## Item 15: Provide access to raw resources in resource-managing classes

- ▶ Cuando creamos un API las clases están hechas para que interactuaremos con ellas, pero hay veces que es necesario permitir al usuario poder acceder al recurso *raw*.
- ▶ Para ello se puede hacer de dos formas: explícita o implícita.
- ▶ La forma explícita será devolver el recurso *raw* mediante la llamada a una función.
- ▶ La forma implícita será sobrecargar el operador de llamada para que devuelva el recurso *raw*.
- ▶ El problema que puede generar es que asociemos ese recurso *raw* a, por ejemplo, una clase hija del mismo. El programa compilará, pero objeto hijo solo tendrá la información del padre, la otra será indefinida. No hay que olvidarse de si liberamos el objeto padre, el objeto hijo se convertirá en un *dangling pointer*.

## Item 16: Use the same form in corresponding uses of `new` and `delete`

- ▶ Si usas `[]` con `new`, debes usar `[]` en su correspondiente `delete`. Si no usas `[]` con `new`, no debes usar `[]` cuando llames a `delete`.

## Item 17: Store `newed` objects in smart pointers in standalone statements I

- ▶ Las funciones pueden requerir el puntero de un objeto en uno de sus parametros. Si este es del tipo *smart\_ptr* podriamos crearlo en la misma llamada.

```
int priority();  
void processWidget(std::shared_ptr<Widget> pw, int priority);  
...  
processWidget(std::shared_ptr<Widget>(new Widget), priority());
```

## Item 17: Store `newed` objects in smart pointers in standalone statements II

- ▶ Hacer esto nos lleva a un comportamiento indefinido por el hecho de que no hay un orden definido por el compilador.
- ▶ El compilador puede generar el código de forma errónea ya que el orden en que ejecuta las llamadas no siempre es el mismo. Un caso correcto sería:
  - ▶ Llama a *priority*.
  - ▶ Ejecuta “new Widget”.
  - ▶ Llama al constructor de *shared\_ptr*.
- ▶ Pero podría darse el caso en que hiciera:
  - ▶ Ejecuta “new Widget”.
  - ▶ Llama a *priority*.
  - ▶ Llama al constructor de *shared\_ptr*.
- ▶ Entonces en el caso de que *priority* causase una excepción se generaría una corrupción de memoria.
- ▶ Por lo tanto, al usar *smart pointers* la solución óptima es almacenar en una variable y usarla para llamar a la función.



## Item 18: Make interfaces easy to use correctly and hard to use incorrectly

- ▶ Para evitar que el usuario use un valor equivocado en las variables, se pueden crear clases o *structs* que ostentan funciones para crearlos. De esa manera no se podrá poner datos erróneos.
- ▶ Usar *shared\_ptr* que soportan destructores personalizados. Con ello prevenimos problemas con DLLs multiplataforma.

## Item 19: Treat class design as type design

- ▶ El diseño de clases es diseño de tipos. Cuando crees un nuevo tipo, considera todas las preguntas de este apartado (Véase en el libro).

## Item 20: Prefer pass-by-reference-to-const to pass-by-value I

- ▶ Cuando usamos *pass-by-value* con una clases esta tiene que llamar al constructor para crearla al inicio de la función y luego al destructor al terminar.
- ▶ Si esta clase es hija de otra o otras clases todas estas tienen que llamar a su constructor y a su destructor.
- ▶ Si alguna de todas estas clases tiene variables no primitivas también habrán que crearse y destruirse.
- ▶ Por ello si usamos *pass-by-reference-to-const* se evitan todas las llamadas a constructores y destructores.

## Item 20: Prefer pass-by-reference-to-const to pass-by-value II

- ▶ Otro error común con al usar *pass-by-value* es usar la clase base como argumento. De este modo, al pasar usando *pass-by-value* una clase hija, esta sera *sliced* y solo se creara una variable con los valores de la clase base y no de la hija.
- ▶ Usa siempre que sea posible *pass-by-reference-to-const* en vez de *pass-by-value*, evitas tanto problemas de *slice* y es más eficiente.
- ▶ Esta regla no se aplica a tipos primitivos ni iteradores y objetos de la STL. Para ellos es más apropiado usar *pass-by-value*.

## Item 21: Don't try to return a reference when you must return an object I

- ▶ Usar *pass-by-reference-to-const* o devolver *const-reference* no es siempre la mejor solución.
- ▶ Si devolvemos una *const-reference* de un objeto creado en el *heap* ya que al salir de la función este se destruirá.
- ▶ Si devolvemos una *const-reference* de un objeto creado en el *stack* el usuario tendrá que acordarse de llamar a su destructor, aun así se pueden dar casos en que sea posible acceder al objeto en cuestión y tendremos *memory leaks*.

## Item 21: Don't try to return a reference when you must return an object II

- ▶ Si devolvemos una *const-reference* de un objeto *static* tendremos problemas, por ejemplo, si lo usamos para sobrecargar el operador de multiplicación, si comparamos dos multiplicaciones que usan la misma función, da igual que valores usemos siempre serán igual, debido a que la variable *static* no cambia. También hace que esa función no sea *thread safe*.
- ▶ Una posible solución es usar el comando *inline* para devolver el objeto como *const-reference*, de ese modo se creará en el *scope* que deseamos.

## Item 22: Declare data members private

- ▶ Todas las variables de una clase han de ser privadas y que estas puedan ser alteradas, si se desea, por funciones *get* y *set*.
- ▶ La finalidad de hacerlas todas privadas sirven para evitar que el usuario las use y obligarle a que use dichas funciones que se han creado.
- ▶ Evitar que el usuario pueda hacer uso de las variables de una clase hace que sea posible eliminarlas o modificarlas en un futuro, ya que de ser publicas o protegidas, los usuarios deberían modificar su código si estas han sido alteradas.