Resumen del libro Effective C++

Elías Serrano

28 de agosto de 2019

Índice I

Índice II

Índice III

Índice IV

Índice V

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

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

- Los **defines** pueden no reflejarse jamas 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. II

- 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
```

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

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

- 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 sera 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 assigment operator** el compilador los creara **públicos** por ti.
- Las funciones **copy constructor** y **copy assigment 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 assigment operator** de esta clase, ya que esta declarado, o en el caso de que sea un elemento simple (p.e int) copiara 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 assigment 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 assigment**, 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 objecto
```

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 assigment**.
- 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 seria:

```
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;
}
```

Item 11: Handle assignment to self in operator= V

- 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= VI

- Si queremos solucionar el problema de **self assigment** 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= VII

```
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);                            // objecto pasado
                                        // intercambia los
                                            datos de
  return *this;                         // *this por el de la
      copia
}
```

Item 11: Handle assignment to self in operator= VIII

- 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 todos 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 la 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 estariamos 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 RAII 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: explicita o implícita.
- La forma explicita 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 sera 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 **new**ed 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 **new**ed 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 seria:
  - 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 optima 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.

Item 23: Prefer non-member non-friend functions to member functions I

- No siempre todas las funciones para manejar una clase han de pertenecer a esta misma. Por ejemplo con la siguiente clase:

```
class WebBrowser {
public:
  ...
  void clearCache();
  void clearHistory();
  void removeCookies();
  ...
};
```

- Crear una función que llame a estas tres seria lo más común, pero la pregunta seria si hacerla parte de la clase o una externa que recibe un **WebBrowser** por parámetro y llama a esas funciones.
- Hacerla parte de la clase sería lo más obvio y lo que se supone que es lo que dicta la programación orientada a objetos, sin embargo no es así, sería entenderlo incorrectamente.

Item 23: Prefer non-member non-friend functions to member functions II

- La programación orientada a objetos dicta que la información debe estar encapsulada lo máximo posible. Las funciones de una clase o una amiga pueden acceder a la información privada, por lo tanto, de querer crear nuestro método, es mucho mejor que sea una función externa la que proceda.
- Esta función puede pertenecer al mismo **namespace** que la clase, no obstante, deberá localizarse en archivos distintos, es así como funciona la STL. El código quedaría de la siguiente forma.

Item 23: Prefer non-member non-friend functions to member functions III

```
// header "webbrowser.h"
class WebBrowser { ... }; ...
  ...  // Funcionalidades básicas
}
// header "webbrowsercookies.h"
namespace WebBrowserStuff {
  ... // Funciones relacionadas con los marcadores
}
// header "webbrowsercookies.h"
namespace WebBrowserStuff {
  ... //Funciones relacionadas con las cookies
}
```

Item 24: Declare non-member functions when type conversions should apply to all parameters

- En el caso de crear un nuevo tipo de datos, si este solo se usa para operar con el mismo no hay problema, pero en el caso de querer operar con otros tipos, por ejemplo, con los primitivos es cuando empiezan los problemas.
- La solución es declarar el operador que se desea utilizar fuera de la clase, debido a que si es parte de la clase solo estos pueden usarlos.

Item 25: Consider support for a non-throwing swap

- Declara la función **swap** cuando **std::swap** vaya a ser ineficiente. Ten en cuenta de que esta no lance excepciones.
- Si ofreces una función miembro **swap**, también tienes que ofrecer una que no sea miembro que llame a la miembro. Para las clases (no **templates**), hazlas parte de **std::swap** también.
- Cuando llamemos a **swap**, utiliza **using** de **std::swap** y usa el método sin el **namespace**, para que así el compilador pueda usar la mejor opción de todas.
- Esta bien añadir especializaciones nuevas a los **templates** de la STD para tipos creados por el usuario, pero nunca intentes añadir algo totalmente nuevo a la STD.

Item 26: Postpone variables definitions as long as possible

- Declarar variables no es gratis ya que supone llamar a su **constructor** cuando se crea y a su **destructor** cuando sale fuera del **scope**.
- En todo momento tienes que retrasar la definición de las variables hasta lo más tarde que puedas.
- En todo momento has de asignar a la variable con información y no crearla y luego asignarle el valor ya que esto incurriría en un coste de asignación extra después de construirla.
- Para bucles es mejor declarar la variables fuera del mismo, solo tienes que recordar que esta variable se destruirá cuando este fuera del **scope** que posiblemente sea al final de la función.
- En el caso que estés en una parte critica del código, es mejor declarar la variable dentro del bucle.

Item 27: Minimize casting

- Tipos de cast que existen:
  - **const_cast** sirve para quitar el **const** del objeto.
  - **dynamic_cast** se usa para realizar `safe downcasting`. Se realiza en tiempo de ejecución.
  - **reinterpret_cast** esta destinado a `casts` de bajo nivel, por ejemplo, convertir un puntero en un int.
  - **static_cast** pueden ser usados para forzar conversiones implícitas o conversiones reversas (un puntero de un objeto base a su derivado).
- Evitar el uso de **cast**. Si el diseño lo necesita, intentad usar una alternativa sin ellos.
- Si es necesario, intentad ocultarlo dentro de una función. Así no hace falta que otros usen **cast** por su cuenta.
- Usar los **casts** de C++ a los de C, debido a que son más reconocibles.

Item 28: Avoid returning “handles” to object internals I

- **handles** son referencias, punteros y iteradores.
- **Pass-by-reference** es mejor que **pass-by-value**. Un error seria intentar hacer lo mismo para los objetos que se devuelven en las funciones.
- El problema de devolver **handles** es que se puede acceder a todo el objeto y podría llegar al siguiente problema.

```
class Rectangle {
  public:
  ...
  Point& UpperLeft() const { return pData->ulhc; }
  Point& LowerRight() const { return pData->lrhc; }
};

Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);  // Un rectangulo const.

rec.UpperLeft().setX(50;)  // ahora rec pasa de (100,0)
                           // a (50, 0).
```

Item 28: Avoid returning “handles” to object internals II

- Una solución seria hacer:

```
class Rectangle {
  public:
  ...
  const Point& UpperLeft() const { return pData->ulhc; }
  const Point& LowerRight() const { return pData->lrhc; }
};
```

- Aunque esto genera otro problema que se llama **dangling handles** que son **handles** que referencia a un objeto que ya no existe.

Item 28: Avoid returning “handles” to object internals III

```
class GUIObject { ... };

// Devuelve un rectangulo por valor constante.
const Rectangle BoundingBox(const GUIObject& obj);

GUIObject *pGUIobj;  // pGUIobj apunta a algún GUIObject.

const Point *pUpperLeft =                    // coger el puntero al
  &(BoundingBox(*pGUIobj).UpperLeft());      // punto superior
                                             // izquierdo de su
                                             //bounding box.
```

- El problema es que se crea un objeto temporal al llamar a **BoundingBox** y cuando se termina el **scope** este se elimina entonces **pUpperLeft** referencia a un objeto eliminado creando asi un **dangling pointer**.

Item 28: Avoid returning “handles” to object internals IV

- Evitad devolver **handles** de objetos internos de una clase. No devolverlos incrementa la encapsulación, ayuda a los mienbros **const** actuar como **const** y minimiza la creación de **dangling handles**.

Item 29: Strive for exception-safe code. I

- Dada la siguente clase:

```
class PrettyMenu {
 public:
  ...
  void changeBackground(std::istream& imgSrc); // Cambia la
  ...                                          // imagen.
 private:
  Mutex mutex;  // Mutex de este objeto.
  Image *bgImage;  // Imagen de fondo actual.
  int imageChanges;  // Número de cambios.
};
```

Item 29: Strive for exception-safe code. II

- Y la siguiente función:

```
void PrettyMenu::changeBackground(std::istream& imgSrc) {
  lock(&mutex);  // Obtener el mutex.
  delete bgImage;  // Borrar la imagen actual.
  ++imageChanges;  // Aumentar el número de cambios.
  bgImage = new Image(imgSrc);  // Poner la nueva imagen.
  unlock(&mutex);  // Liberar el mutex.
}
```

- Podemos ver que de darse una excepción la cosa se puede poner fea.

Item 29: Strive for exception-safe code. III

- Por lo tanto, si buscamos **exception-safety** hay dos aproximaciones:
  - **Leak no resources**, si se da una excepción crean la imagen nueva, el **mutex** jamas se liberará.
  - **No permitir que las estructuras se hagan corruptas**, si se da una excepción en el constructor, el puntero apuntara a un objeto eliminado y el incremento se habrá hecho.
- Solucionar el primer problema es fácil, usando RAII visto en el item 1, el **mutex** se liberará cuando salga del **scope** usando los **locks** del item 2:

```
void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock ml(&mutex);
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

- Menos código mejor, reduce la frequencia de que algo vaya mal.

Item 29: Strive for exception-safe code. IV

- Referente al problema segundo problema hay tres formas de solucionar de abarcarlo:
  - **La garantia básica**, la función garantiza que de darse un excepción todo quedará en un estado valido. Por ejemplo:

```
class PrettyMenu {
  ...
  std::shared_ptr<Image> bgImage;  // usamos shared_ptr.
  ...
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock ml(&mutex);
  bgImage.reset(new Image(imgSrc));  // Cambiamos el
                                     // punteor por el
                                     // nuevo.
  ++imageChanges;
}
```

  De esta forma, solo si se consigue crear el nuevo objeto se cambiara el actual.

Item 29: Strive for exception-safe code. V

- **La garantia fuerte**, la función garantiza que si se da una excepción el estado quedará igual. Por ejemplo, si cambiamos la clase para que use una estructura con la imagen y el número de cambios:

```
struct PMImpl {
  std::shared_ptr<Image> bgImage;
  int imageChanges;
};
class PrettyMenu {
  ...
 private:
  Mutex mutex;
  std::shared_ptr<PMImpl> pImpl;
};
```

Item 29: Strive for exception-safe code. VI

Luego, con el uso de **swap** podemos garantizar la **garantia fuerte**:

```
void PrettyMenu::changeBackground(std::istream& imgSrc) {
  using std::swap;
  Lock ml(&mutex);
  // Copiamos el objeto actual.
  std::shared_ptr<PMImpl> pNew(new PMImpl(*pImpl));
  // Cambiamos el puntero por el objeto nuevo.
  pNew->bgImage.reset(new Image(imgSrc));
  ++pNew->imageChanges;  // Incrementamos.
  swap(pImpl, pNew);  // Intercambiamos los objetos.
}
```

De esta forma, solo se usará la nueva imagen y se incrementará si se consigue crear el objeto nuevo con la imagen actual, cambiar por la imagen nueva y el **swap** se efectua con éxito.

Item 29: Strive for exception-safe code. VII

- **La garantia nothrow** promete que la función nunca lanzará ninguna excepción porqué hacen siempre lo que promete. Esto no quiere decir que no haya excepciones, sino que de darse alguna es un problema muy serio.

Item 30: Understand the ins and outs of inlining.

- **Inline** es un petición al compilador, no una instrucción.
- Sirve para decirle al compilador que reemplace la llamada a la función por el código de la misma.
- Si se usa demasiado, el código aumentara, lo que lleva a **additional paging**, se reduce el **instruction cache hit rate** y las penalizaciones en rendimiento que acarrea todo eso.
- Si se usa para funciones pequeñas muy utilizadas, el código que se genera es menor y aumenta el **instruction cache hit rate**.
- Hay dos formas de hacer las funciones **inline**:
  - Declarando la función en la cabecera.
  - Usando el comando **inline**.
- Hay que tener cuidado con declarar **inline** las funciones **template** solo por el hecho de que se declaran en las cabeceras. Solo hay que hacerlo cuando se crea necesario.

Item 31: Minimize compilation dependencies between files.

- El compilador necesita saber el tamaño de los objetos que se usan en tiempo de compilación. Por lo tanto, algo común es incluir las cabeceras debe dichos objetos.
- El problema de incluir estas cabeceras es que un cambio en ellas te hara tener que recompilar todas las clases donde se usen.
- Hay tres simples estrategias de diseño para solucionar este problema:
  1. Evitar usar objetos cuando referencias o punteros son suficientes.
  2. Depender en la declaración de las clases y no en la definición de las clases.
  3. Usar cabeceras distintas para la declaración y definición de las clases. Este se conoce como **pimpl idiom**. Aun así, el uso de esta estrategia reduce la velocidad en ejecución y aumenta el uso de memoria por cada objeto que lo use.

Item 32: Make sure public inheritance models “is-a.” I

- Todo lo que se aplica a las clases base también se aplica a las clases derivadas, porqué toda clase derivada es una clase base.
- Por lo tanto, si creasemos la clase pájaro con la función para volar y luego definiesemos la clase pingüino que hereda de pájaro de esta forma:

```
class Bird {
  virtual void fly();
  ...
};
class Penguin: public Bird {
  ...
};
```

- Como todo lo que se aplica en pájaro se aplica en pingüino, tendremos que estos son capaces de volar, cuando no lo son.

Item 32: Make sure public inheritance models “is-a.” II

- Una posible solución seria crear una clase “animal volador” de la siguiente forma:

```
class Bird {
  ...  // No fly function is declared.
};
class FlyingAnimal: public Bird {
  virtual void fly();
  ...
}
class Penguin: public Bird {
  ... // No fly function is declared.
};
```

- De este modo, los pingüinos no son capaces de volar.
- O tambíen se podria sobreescribir el método volar para que devuelva un error si intentas hacemos que el pingüino vuele.

Item 33: Avoid hiding inherited names. I

- Para ponernos en contexto, cuando declaramos una variable, esta tiene preferencia antes otras con el mismo nombre fuera del **scope**, por ejemplo:

```
int x;  // Global variable.
void someFunc() {
  double x;  // Local variable.
  std::cin << x;  // Read a new value
}                 // for local x.
```

- Este mismo comportamiento sucede en la herencia.
- Si tenemos las siguientes clases:

Item 33: Avoid hiding inherited names. II

```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf2();
  void mf3();
  ...
};
class Derived: public Base {
public:
  virtual void mf1();
  void mf4() {
    ...
    mf2();
    ...
  }
  ...
};
```

Item 33: Avoid hiding inherited names. III

- Cuando el compilador vea la llamada a la función mf2 en mf4 tendrá que buscar a que función hace referencia. Seguirá el siguiente procedimiento:
  1. Buscará en el **scope** de mf4.
  2. Mirara en la clase Derived.
  3. Por último, ira a la clase Base.
- En el primer lugar donde la encuentre es el que usará.
- Entonces, si tenemos las siguientes clases:

Item 33: Avoid hiding inherited names. IV

```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
  ...
};
class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```

Item 33: Avoid hiding inherited names. V

- Si ahora ejecutasemos el siguientes código tendriamos que:

```
Derived d; int x;
...
d.mf1();   // Bien, llama aDerived::mf1.
d.mf1(x);  // Error, Derived::mf1() oculta Base::mf1.

d.mf2();   // Bien, llama a Base::mf2.
d.mf3();   // Bien, llama a Derived::mf3.
d.mf3(x);  // Error, Derived::mf3 oculta Base::mf3.
```

- Para poder solucionar este problema tenemos que darle al compilador la información de donde buscar. Esto se consigue usando **using** de la siguiente forma:

Item 33: Avoid hiding inherited names. VI

```
class Derived: public Base {
public:
  using Base::mf1;
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```

- En el caso de que solo queramos **overload** unas pocas funciones que compartan nombre, tendremos que usar el comando **using** para que el compilador sea capaz de encontrar la función correcta que no hemos **overload**.

Item 34: Differentiate between inheritance of interface and inheritance of implementation.

- Heredar de una interfaz es diferente que heredad de una implentación. En la herencia pública, las clases derivadas siempre heredan las interfaces de las clases base.
- Las funciones virtuales puras especifican herencia de interfaz solamente.
- Funciones virtuales simples (inpuras) especifican herencia de interfaz más herencia de una implementación por defecto.
- Funciones no-virtuales espeficican herencia de interfaz más herencia de una implementación.

Item 35: Consider alternatives to virtual functions. I

- Dada la siguiete clase:

```
class GameCharacter {
public:
  virtual int healthValue() const; // Devuelve la vida del
  ...                              // personaje.
};
```

- Las diferentes estrategias para no usar el convencional diseño orientado a objetos serian:

Item 35: Consider alternatives to virtual functions. II

1. The Template Method Pattern via the Non-Virtual Interface Idiom.
   - En este diseño tendremos la función healthValue publica y sin ser virtual y la función que hará el trabajo y podrá ser sobrecargada será una privada llamada doHealthValue como podemos ver en el siguiente código:

```
class GameCharacter {
public:
  int healthValue() const {
    ...
    int retVal = doHealthValue();
    ...
    return retVal;
  }
...
private:
  virtual int doHealthValue() const { ... }
};
```

   - En este diseño ganamos la habilidad de poder hacer algo antes y después de llamar a la función virtual.
   - Aunque podamos cambiar el comportamiento de doHealthValue, la clase base se reserva el derecho de llamar a la clase virtual.

Item 35: Consider alternatives to virtual functions. III

2. The Strategy Pattern via Function Pointers.
   - La idea es usar punteros a funciones que tengan el comportamiento que se desea, por ejemplo:

```
class GameCharacter;
// Algoritmo por defecto.
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(
    HealthCalcFunc hcf = defaultHealthCalc) :
    healthFunc(hcf ) {}

  int healthValue() const { return healthFunc(*this); }
  ...
private:
  HealthCalcFunc healthFunc;
};
```

Item 35: Consider alternatives to virtual functions. IV

- Esto nos ofrece la flexibilidad para:

1. Diferentes instancias del mismo objeto pueden usar funciones distintas:

```
class EvilBadGuy: public GameCharacter {
public:
  explicit EvilBadGuy(
    HealthCalcFunc hcf = defaultHealthCalc) :
    GameCharacter(hcf ) { ... }
  ...
};

int loseHealthQuickly(const GameCharacter&);
int loseHealthSlowly(const GameCharacter&);

EvilBadGuy ebg1(loseHealthQuickly);
EvilBadGuy ebg2(loseHealthSlowly);
```

2. La función puede cambiarse en tiempo de ejecución.

Item 35: Consider alternatives to virtual functions. V

3. The Strategy Pattern via stl::function.
   - Para este diseño usaremos el mismo código que en punto anterior cambiando el **typedef** por:

```
typedef std::tr1::function<int (const GameCharacter&)>
    HealthCalcFunc;
```

   - Ahora usamos stl::function en vez de un puntero normal a una función.

Item 35: Consider alternatives to virtual functions. VI

- Por lo tanto, si tenemos estas clases:

```
short calcHealth(const GameCharacter&);
struct HealthCalculator {
  int operator()(const GameCharacter&) const { ... }
};

class GameLevel {
public:
  float health(const GameCharacter&) const;
  ...
};

class EvilBadGuy: public GameCharacter { ... };
class EyeCandyCharacter: public GameCharacter { ... };
```

- Nos permite poder hacer lo siguiente:

Item 35: Consider alternatives to virtual functions. VII

```
EvilBadGuy ebg1(calcHealth); // Usando la función.
EyeCandyCharacter ecc1(
  HealthCalculator()); // Usando la función del objeto

GameLevel currentLevel;
...
EvilBadGuy ebg2(
  std::bind(&GameLevel::health,
                currentLevel,
                _1));
```

- stl::function nos da mucha versatibilidad. Podemos pasar una función, la función de un objeto o usar std::bind.
- std::bind lo que hace es “anidar” la funcion `health` del objeto `currentLevel` para hacer el calculo.

Item 35: Consider alternatives to virtual functions. VIII

4. The “Classic” Strategy Pattern.
   - Sino deseamos usar funcionalidades de C++ y quedarnos con un enfoque clásico, podemos hacer que el calculo sea una clase base la cual pueda ser heredada por otras que cambien la función. Por ejemplo:

Item 35: Consider alternatives to virtual functions. IX

```
class GameCharacter;
class HealthCalcFunc {
public:
  ...
  virtual int calc(const GameCharacter& gc) const {...}
  ...
};
HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
  explicit GameCharacter(
    HealthCalcFunc *phcf = &defaultHealthCalc)
    : pHealthCalc(phcf ) {}
int healthValue() const {
  return pHealthCalc->calc(*this);
}
private:
  HealthCalcFunc *pHealthCalc;
};
```

Item 36: Never redefine an inherited non-virtual function.

- Si una clase derivada usa el mismo nombre de una función base que no es virtual, esta ocultara la de la clase base y se llamará a la que esta en la derivada:

```
class B {
public:
  void mf();
};

class D: public B {
public:
  void mf();
};

D x;
B* pB = &x;
D* pD = &x;

pB->mf();  // Llama a B:mf
pD->mf();  // Llama a D:mf
```

Item 37: Never redefine a functions inherited default parameter value.

- Las funciones virtuales son **dynamically bound**, pero los parametros por defecto son **statically bound**.
- Si sobrecargamos una función con un parametros por defecto de la base y cambiamos ese valor en la derivada, el valor por defecto a usar siempre será el de la clase base:

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };
  virtual void draw(ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape { public:
  virtual void draw(ShapeColor color = Green) const;
};

Shape *ps;
Shape *pr = new Rectangle;

ps = pr; // El tipo dinamica de ps es Rectangle.
pr->draw(Shape::Red); // Llama a Rectangle::draw(Shape::Red)
```

Item 38: Model has-a or is-implemented-in-terms- of through composition.

- **Composition** es la relación entre los tipos de un objeto que surgen cuando los objetos de un tipo contienen objetos de otro.
- En el dominio de la aplicación, **composition** se traduce a **has-a**.
- En el dominio de la implementación, se refiere a **is-implemented-in-terms-of**.