UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Introducción a la programación

Apunte extra
Templates, clases friend, métodos friend y sobrecarga de operadores

CONTENIDOS

1. Herramientas varias: como navaja suiza

La Programación Orientada a Objetos (POO) tiene tres pilares básicos: el objeto con su encapsulamiento, la herencia y el polimorfismo. Estos temas fueron desarrollados en las unidades precedentes. Sin embargo, esta división no quita que existan otras herramientas, técnicas o usos que nos permitan mejorar nuestro código, haciéndolo más robusto, más prolijo o, en definitiva, más útil.

En esta unidad vamos a conocer algunas de estas herramientas.

Organización de archivos

La organización de archivos puede no parecer una herramienta, ya que no contribuye directamente al desarrollo de software. Sin embargo, tener los archivos bien clasificados y distribuidos nos ayuda a esquematizarnos mejor y a tener un panorama más claro de lo que debemos hacer.

Hasta ahora, cuando construíamos una clase, lo hacíamos declarando primero la cabecera y luego la implementación de los métodos en un mismo archivo *cpp;* incluso teníamos el main en ese archivo.

C++ propone la siguiente organización para los archivos de una clase:

Archivos ****.h****:* contienen la *declaración* de la clase, esto es, lo que hemos hecho hasta ahora al principio del archivo.

Archivos ****.cpp****:* contienen la *implementación* de los métodos declarados en la cabecera, es decir, lo que hacíamos a continuación de la declaración de la clase.

Tanto los archivos *.cpp* como *.h* deben tener el nombre de la clase.

Esta división de archivos no tiene que ver sólo con una cuestión organizativa, sino que interfieren funciones del compilador que aprovechan las cabeceras para manejar propiedades de preprocesamiento. Si bien aquí no ahondaremos en estas cuestiones, aprovecharemos la división para darle un poco de claridad y organización a nuestro código.

Tomemos como ejemplo el caso de una clase *vehiculo* que tiene un método *conducir*:

vehiculo.h

```
#ifndef VEHICULO_H
#define VEHICULO_H

class vehiculo {

public:

    vehiculo();
    ~vehiculo();
    void conducir();

};

#endif
```

Script 1

vehiculo.cpp

```
#include <iostream>
#include "vehiculo.h"

using namespace std;

vehiculo::vehiculo() {
    // constructor
}

vehiculo::~vehiculo() {
    // destructor
}

void vehiculo::conducir(){

    cout<< "Manejando mi vehiculo"<<endl;
}
```

Script 2

vehiculo_test.cpp

```
#include <iostream>
#include "vehiculo.h"

using namespace std;

int main (int argc, char *argv[]) {

    vehiculo V;
    V.conducir();

    return 0;
}
```

Script 3

Notemos que tanto el archivo que contiene el main, como la implementación de los métodos, incluyen al archivo *vehiculo.h.*

Un par de aclaraciones:

En vehiculo.h, todo se declara dentro de la etiqueta *ifndef* y cierra con una *endif*, esto es, una herramienta de C++ que nos protege del duplicado de código, en caso de que agreguemos más de una vez la misma librería. Lo que hace el compilador es crear el objeto vehiculo_h; luego, si lo vuelve a leer porque otra librería también lo incluye, lo pasa por alto y utiliza el creado por primera vez.

En C++, la instrucción *include* puede hacerse tanto con los símbolos de mayor y menor (< y >), como con las comillas dobles (“”), como ocurre en *vehiculo.cpp.* La utilización de estos símbolos no es indistinta. Si empleamos las comillas, el compilador buscará el archivo en el directorio actual de trabajo y, de no encontrarlo, lo hará por defecto en el directorio, que suele denominarse *include,* donde se alojan los *headers (archivos .h).* Por el contrario, si utilizamos los símbolos de menor y mayor, el proceso de búsqueda será al revés.

Si tuviésemos otra clase, habría otro nuevo grupo de archivos. La cuestión se vuelve más compleja cuando contamos con clases con dependencia, por ejemplo, una herencia.

Veamos el caso de la clase *automotor,* que hereda de vehiculo:

automotor.h

```
#include "vehiculo.h"

#ifndef AUTOMOTOR_H
#define AUTOMOTOR_H

class automotor : public vehiculo{
  public:
      automotor();
      ~automotor();
      void abrir_puertas();
};

#endif
```

Script 4

automotor.cpp

```
#include <iostream>
#include "automotor.h"

using namespace std;

automotor::automotor() {
    // constructor
}

automotor::~automotor() {
    // destructor
}

void automotor::abrir_puertas(){
    conducir();
    cout<< "Abriendo las puertas"<<endl;
}
```

Script 5

Y *vehiculo_test* queda:

vehiculo_test.cpp

```
#include <iostream>
#include "automotor.h"

using namespace std;

int main (int argc, char *argv[]) {

    vehiculo V;
    V.conducir();
    automotor A;
    A.abrir_puertas();

    return 0;
}
```

Script 6

Como vemos, en ningún momento incluimos a vehiculo.h, a pesar de que lo usamos. Esto se debe a que el mismo ya está comprendido en *automotor.h.*

De todos modos, si lo hubiésemos incorporado, no habría ocurrido nada, ya que –como mencionamos– el ifndef nos protege de las múltiples declaraciones. Esto es lógico en la medida en que, cuando creamos una clase que tiene múltiples herencias de uso privado y otros elementos que no deseamos dejarlos a la vista del usuario, sólo incluiremos al código la clase resultante, a saber, la que tiene todo elaborado como para llevar a cabo lo que necesita.

Por ejemplo, la clase *iostream* que siempre incluimos para hacer uso de la salida y entrada de flujo tiene muchas dependencias en los niveles superiores. Sin embargo, para nosotros es completamente trasparente, lo cual es un alivio, ya que si incluyéramos cada clase dependiente de todas las que incorporamos para trabajar, tendríamos cientos y cientos de líneas de *include*.

Sobrecarga de operadores

La sobrecarga de operadores es una de las herramientas más poderosas que nos brinda C++ para mejorar la interacción en nuestro código.

Si bien es un tema extenso para explicarlo detalladamente, intentaremos aproximarnos al mismo con ejemplos simples.

Como su nombre lo indica, la *sobrecarga de operadores* consiste –básicamente– en modificar la funcionalidad de un operador para darle una utilidad distinta, o en definirle una utilidad que antes no tenía.

Por ejemplo, si queremos sumar dos enteros *a* y *b*, hacemos *a+b*. Ahora bien, si tenemos un juego donde hay que controlar cientos de bacterias que deben comer a un organismo y que pueden fusionarse para formar una sola bacteria, lo ideal sería hacer:

nueva_bacteria = bacteria1+bacteria2;

Donde la nueva bacteria resultante contiene la suma de los atributos de bacteria1 y bacteria2. Más aún, podríamos sumar muchas bacterias.

De la misma manera, sería posible crear un juego donde hay que controlar a un pulpo y podamos acceder a cada uno de los tentáculos con el operador corchete. Así:

Pulpo[1] para el tentáculo 1

Pulpo[2] para el tentáculo 2

Etc.

Esta modificación de la función de un operador se denomina sobrecarga de operadores y es una de las características que más se extraña de C++ cuando se programa en otro lenguaje.

Los operadores se clasifican en tres categorías no exclusivas, a saber:

- *De acuerdo al estado resultante del objeto:* al ejecutar un operador, el mismo puede afectar el estado de un objeto o retornar un valor sin alterar los valores del mismo.
- *De acuerdo a si se ejecuta pre o post al objeto:* por ejemplo, objeto++ es *post* al objeto, mientras que ++objeto es *pre* al objeto. C++ sólo reconoce a los operadores ++ y – como candidatos para post objetos.
- *De acuerdo a la cantidad de operandos que afecta:* Un operador puede ser unario si afecta a un solo objeto, o binario, si afecta a varios. Así, *&ObjetoA* es unario, mientras que *objetoA + objetoB* es binario.

A fin de tornar más didáctica la explicación, utilizaremos esta última categorización para desarrollar el tema.

La siguiente lista contiene los operadores que C++ permite sobrecargar:

```
+    -    *    /    %    ^    &    |    ~    !
=    <    >    +=   -=   *=   /=   %=   ^=
&=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   –    ->*  ,    ->   []   ()
```

Sobrecarga de operadores unarios

Los operadores unarios que C++ permite sobrecargar son:

```
+    -    *    &    ~    !    ++   –
```

La sintaxis de la sobrecarga de operadores unarios es la siguiente:

<tipo_retorno> operator **(o)** ( )

Donde **(o)** es, precisamente, el operador a sobrecargar. Éste puede ser cualquiera de los que nombramos anteriormente.

A continuación, crearemos una clase que nos servirá de ejemplo para explicar la sobrecarga de operadores.

Hagamos, entonces, efectiva nuestra clase *bacteria.* La misma tiene algunos atributos como *vida* (que es la vida que tiene), *diámetro y rango* (rango de daño).

En primer lugar, definimos los operadores unarios:

++: aumenta la vida en uno de nuestra bacteria (tanto el pre como el post objeto).

–: disminuye la vida en uno de nuestra bacteria (tanto el pre como el post objeto).

~: nos informa sobre el rango de nuestra bacteria.

+: devuelve una bacteria con el doble de vida.

-: devuelve una referencia a un objeto bacteria con el doble de vida.

A la hora de programar un operador debemos determinar con precisión qué deseamos del mismo. Para esto, quizás, resulte útil clasificarlo.

Ahora bien, como sabemos que todos esos operadores son unarios, ya estamos en condiciones de observar cómo afectan el estado del objeto.

Si al hacer +Mi_bacteria estamos retornando un objeto bacteria con el doble de vida de la bacteria que está efectuando el operador, no estaremos modificando en lo absoluto el estado del objeto. Lo mismo ocurre si hacemos ~Mi_bacteria. Sin embargo, al hacer Mi_bacteria++ o Mi_bacteria—, modificaremos el estado mismo del objeto Mi_bacteria, similar a lo que sucede con ++Mi_bacteria y –Mi_bacteria.

A continuación, debemos analizar qué retorna y qué toma cada operador como parámetro:

- *bacteria++:* estaría retornando un elemento bacteria y tomando como parámetro un entero (la sintaxis de la sobrecarga de este método es única. C++ no admite otro tipo de dato que no sea *int* para la sobrecarga de los operadores ++ y – post objeto).
- *bacteria—:* se repite el caso de bacteria++.
- *++bacteria:* no toma ningún parámetro y retorna un elemento bacteria.
- *–bacteria:* similar al caso de –bacteria.
- *~bacteria:* retorna el atributo del rango. No toma ningún parámetro.

- *+bacteria:* retorna un elemento bacteria. No toma ningún parámetro.
- *-bacteria:* le ocurre lo mismo que a +bacteria, pero retorna una referencia a un elemento bacteria (*bacteria).

Veamos, entonces, cómo queda la cabecera de nuestra clase bacteria:

bacteria.h

```
using namespace std;

class bacteria {
public:

    bacteria(int,int);

    /* Operadores unarios */

    int operator~ ();           //retorna rango
    bacteria operator+ ();      //retorna un objeto con doble de vida
    bacteria* operator- ();     //retorna referencia a un objeto con
doble de vida

    bacteria operator++ (int);  //postincremento
    bacteria operator-- (int);  //postdecremento
    bacteria operator++ ();     //preincremento
    bacteria operator-- ();     //predecremento

    int ver_vida();

private:
    int vida;
    int diametro;
    int rango;
};
```

Script 7

A continuación, es posible observar la declaración de los métodos y cómo realizamos la sobrecarga de los operadores:

bacteria.cpp

```
#include<iostream>

bacteria::bacteria(int v= 100,int r = 1){
    vida = v;
    rango = r;
    diametro = 50;

}
int bacteria::ver_vida(){return vida;}

int bacteria::operator~(){
    return rango;
}

bacteria bacteria::operator+(){
    bacteria bc(vida*2);
    return (bc);
}
bacteria* bacteria::operator-(){
    bacteria *bc = new bacteria(vida*2);
    return (bc);
}

bacteria bacteria::operator++(int){
    bacteria bc(vida,rango);
```

```
    vida++;
    return bc;
}
bacteria bacteria::operator--(int){
    bacteria bc(vida,rango);
    vida--;
    return bc;
}
bacteria bacteria::operator--(){
    vida--;
    return *this;
}
bacteria bacteria::operator++(){
    vida++;
    return *this;
}
```

Script 8

Finalmente, probamos nuestros operadores en el *main.cpp:*

main.cpp

```
int main (int argc, char *argv[]) {

    bacteria B;
    cout<< ~B <<endl;
    bacteria C = +B;
    cout<< C.ver_vida() <<endl;
    bacteria *D = -B;
    cout<< D->ver_vida()<<endl;
    cout<< (B++).ver_vida()<<endl;
    cout<< B.ver_vida()<<endl;
    cout<< (++B).ver_vida()<<endl;
    cout<< B.ver_vida()<<endl;

    return 0;
}
```

Script 9

Hasta aquí los parámetros de los operadores eran unarios. A continuación, veremos cómo se comportan los operadores de varios parámetros.

Sobrecarga de operadores binarios

Como lo indicamos anteriormente, los operadores binarios trabajan con dos parámetros.

Los operadores binarios que C++ permite sobrecargar son:

+ - * / % ^ & | = <

> += -= *= /= %= ^= &= |=

<< >> >>= <<= == != <= >=

&& || ->* , -> [] ()

La sintaxis de la sobrecarga de operadores binarios es la siguiente:

<tipo_retorno> operator **(o)** (parámetro)

Dónde **(o)** es el operador a sobrecargar.

Retomando el ejemplo de la clase bacteria, definiremos sobre la misma los siguientes operadores binarios:

+: suma dos bacterias y retorna una bacteria con el doble de vida y rango.

(v): la bacteria recibe *v* puntos de daño.

Los operadores binarios reciben un parámetro; el otro parámetro es el objeto propio. Para decirlo de otro modo, uno de los operandos es el residente de la acción, mientras que el otro operando es un parámetro.

Detallemos, entonces, cómo son los métodos de los operadores:

- *bacteria+bacteria*: retoma un elemento bacteria y recibe otro elemento bacteria como parámetro.
- *bacteria(valor):* retorna un elemento bacteria y toma un entero como parámetro.

Ahora, veamos cómo queda la cabecera de nuestra clase bacteria con los operadores binarios agregados:

bacteria.h

```
using namespace std;


class bacteria {
public:

    bacteria(int,int);

    /* Operadores unarios */

    int operator~ ();           //retorna rango
    bacteria operator+ ();      //retorna un objeto con doble de vida
    bacteria* operator- ();     //retorna referencia a un objeto con
doble de vida

    bacteria operator++ (int);  //postincremento
    bacteria operator-- (int);  //postdecremento
    bacteria operator++ ();     //preincremento
    bacteria operator-- ();     //predecremento

    /* Operadores binarios */

    bacteria operator+(bacteria); //suma entre bacterias
    bacteria operator()(int);       //recibir danio de bacteria

    int ver_vida();

private:
    int vida;
    int diametro;
    int rango;
};
```

Script 10

bacteria.cpp

```
#include<iostream>

bacteria::bacteria(int v= 100,int r = 1){
    vida = v;
    rango = r;
    diametro = 50;
}
int bacteria::ver_vida(){return vida;}

int bacteria::operator~(){
    return rango;
}

bacteria bacteria::operator+(){
    bacteria bc(vida*2);
    return (bc);
}

bacteria* bacteria::operator-(){
    bacteria *bc = new bacteria(vida*2);
    return (bc);
}

bacteria bacteria::operator++(int){

    bacteria bc(vida,rango);
    vida++;
    return bc;
}

bacteria bacteria::operator--(int){

    bacteria bc(vida,rango);
    vida--;
    return bc;
}

bacteria bacteria::operator--(){

    vida--;
    return *this;
}
bacteria bacteria::operator++(){

    vida++;
    return *this;
}

bacteria bacteria::operator+(bacteria b1){
    bacteria BC(vida+b1.ver_vida(),rango+(~b1));
    return BC;
}
bacteria bacteria::operator()(int v){
    vida-=v;
    return (bacteria(vida,rango));
}
```

Script 11

main.cpp

```
int main (int argc, char *argv[]) {

    bacteria B;
    cout<< ~B <<endl;
    bacteria C = +B;
    cout<< C.ver_vida() <<endl;
    bacteria *D = -B;
```

```
    cout<< D->ver_vida()<<endl;
    cout<< (B++).ver_vida()<<endl;
    cout<< B.ver_vida()<<endl;
    cout<< (++B).ver_vida()<<endl;
    cout<< B.ver_vida()<<endl;
    bacteria E = B+C;
    cout<< E.ver_vida() <<endl;
    cout<< ~E<<endl;
    E(100);
    cout<< E.ver_vida()<<endl;
    return 0; }
```

Script 12

Cabe destacar que cuando sobrecargamos un operador, estamos eliminando la definición por defecto que el mismo pueda llegar a tener. Por ejemplo, si modificamos el operador = que utilizamos abiertamente en nuestros ejemplos, perderemos su función original.

Por último, es importante remarcar que C++ también permite recargar los operadores de forma global. La sintaxis es la misma, sólo que no se implementa como clase, sino como funciones globales. Además, en lugar de tener un objeto residente y tomar a otro como parámetro, toma dos parámetros. Por ejemplo, para recargar el operador *(>)* de dos clases vehiculo con un parámetro *kilometraje,* hacemos:

```
vehiculo operator > (vehiculo V1, vehiculo V2){
    if (V1.kilometraje > V2.kilometraje)
        return V1;
    return V2;
}
```

Script 13

Como vemos, la sobrecarga de operadores resulta fascinante, ya que nos permite reducir enormemente la complejidad y cantidad de código necesario para realizar algunas acciones. Si bien no es imprescindible utilizar operadores en nuestras clases, los mismos aportan una interfaz muy agradable y nos brindan claridad a la hora de programar.

Templates

Cuando necesitamos ejecutar una misma acción pero con distintos tipos de datos, C++ nos permite sobrecargar funciones o métodos que se diferencian en el tipo de valores que trabajan. Esta es una excelente solución. Sin embargo, a veces resulta muy incómodo hacer semejante duplicación de código sólo para satisfacer a todos los tipos de datos que necesitan usarlo.

C++ permite implementar un tipo de dato *comodín* que es útil para realizar un solo código que sea aplicable a infinitos tipos de datos, siempre y cuando los mismos se adapten a nuestro algoritmo. Es decir, no podemos pedir que se ejecute un método *volar* a la clase *caballo.*

Los *templates* o *plantillas* constituyen, justamente, ese tipo de dato comodín que luego tomará un valor en tiempo de ejecución.

Las plantillas se definen de la siguiente manera:

template <class tipo>

Donde *tipo* es la plantilla, que luego será reemplazado como un tipo de dato particular en tiempo de ejecución. La definición de la plantilla se coloca

previamente a la línea que la utiliza. Cada plantilla sirve como un tipo de dato. Por lo tanto, si necesitásemos varios datos, tendríamos que definir varias plantillas.

Planteemos el siguiente ejemplo:

En una clase *comarca* hay tres tipos de personajes: ogros, elfos y hobbits. Todos tienen el atributo vida, pero en cada clase es de un tipo diferente. La clase comarca implementa un método para comparar qué personaje es más grande.

Volveremos a unificar las clases para no tener tantos archivos:

```
#include<iostream>

using namespace std;

class ogro {
public:
    ogro();
    int vida;
};

class hobbit {
public:
    hobbit();
    float vida;
};

class elfo {
public:
    elfo();
    double vida;
};

ogro::ogro(){ vida=20;}
elfo::elfo(){vida = 20.000;}
hobbit::hobbit(){vida=20.0;}


class comarca{

    public:

    ogro Ogro_1;
    ogro Ogro_2;
    elfo Elfo_1;
    elfo Elfo_2;
    hobbit Hobbit_1;
    hobbit Hobbit_2;

    template <class Tipo1>
    int mas_grande(Tipo1,Tipo1);
};

template <class Tipo1>

int comarca::mas_grande(Tipo1 t1, Tipo1 t2){

    return ((t1>t2)? 1:2);

}

int main (int argc, char *argv[]) {


    comarca Mi_comarca;
    Mi_comarca.Ogro_1.vida-=5;
    Mi_comarca.Ogro_2.vida+=10;
    Mi_comarca.Elfo_1.vida+=10;
    Mi_comarca.Elfo_2.vida-=10;
```

```
cout <<
Mi_comarca.mas_grande(Mi_comarca.Ogro_1.vida,Mi_comarca.Ogro_2.vida)<<end
l;
cout <<
Mi_comarca.mas_grande(Mi_comarca.Elfo_1.vida,Mi_comarca.Elfo_2.vida)<<end
l;

    return 0;
}
```

Script 14

Una plantilla se define en la declaración del método y otra en forma previa a la implementación. Como mencionamos anteriormente, Tipo1 (la plantilla definida) funciona como un único tipo por vez. Si quisiéramos comparar un ogro contra un elfo (int contra double), tendríamos que hacer dos plantillas en lugar de una.

El uso de las plantillas no se limita a los tipos comunes de datos, sino que su verdadera virtud se encuentra en la capacidad de ser comodín de cualquier tipo de dato, incluso de clases definidas por el usuario.

Planteemos, por ejemplo, un método *combate* que devuelva el objeto ganador del enfrentamiento entre dos objetos. Para decidir cuál es el objeto victorioso, nos fijamos en aquel que tiene mayor vida. La gran propiedad de las plantillas radica en la posibilidad de efectuar un sólo método y aprovechar que todas las clases tienen el atributo vida.

Veamos entonces cómo queda nuestra clase comarca:

```
#include<iostream>
using namespace std;

class ogro {
public:
    ogro();
    int vida;
};

class hobbit {
public:
    hobbit();
    float vida;
};

class elfo {
public:
    elfo();
    double vida;
};

ogro::ogro(){ vida=20;}
elfo::elfo(){vida = 20.000;}
hobbit::hobbit(){vida=20.0;}


class comarca{

    public:

    ogro Ogro_1;
    ogro Ogro_2;
    elfo Elfo_1;
    elfo Elfo_2;
    hobbit Hobbit_1;
    hobbit Hobbit_2;

    template <class Tipo1>
```

```
    int mas_grande(Tipo1,Tipo1);

    template <class Tipo1>
    Tipo1 combate(Tipo1,Tipo1);


};
/* nos devuelve 1 si el objeto mas grande es el primero
 caso contrario devuelve, compara el atributo vida */

template <class Tipo1>
int comarca::mas_grande(Tipo1 t1, Tipo1 t2){
    return ((t1>t2)? 1:2);
}

/* la clase combate devuelve un objeto y permite comparar 2 objetos
comodines */

template <class Tipo1>
Tipo1 comarca::combate(Tipo1 t1, Tipo1 t2){
    return ((t1.vida>t2.vida)? t1:t2);
}


int main (int argc, char *argv[]) {


    comarca Mi_comarca;
    Mi_comarca.Ogro_1.vida-=5;
    Mi_comarca.Ogro_2.vida+=10;
    Mi_comarca.Elfo_1.vida+=10;
    Mi_comarca.Elfo_2.vida-=10;

    cout <<
Mi_comarca.mas_grande(Mi_comarca.Ogro_1.vida,Mi_comarca.Ogro_2.vida)<<endl
;
    cout <<
Mi_comarca.mas_grande(Mi_comarca.Elfo_1.vida,Mi_comarca.Elfo_2.vida)<<endl
;

    ogro Ogro_ganador =
Mi_comarca.combate(Mi_comarca.Ogro_1,Mi_comarca.Ogro_2);
    elfo Elfo_ganador =
Mi_comarca.combate(Mi_comarca.Elfo_1,Mi_comarca.Elfo_2);

    return 0;
}
```

Script 15

Como última acotación, y para reforzar la potencia de esta herramienta, agregaremos que C++ no limita el uso de plantillas a las clases, sino que también permite utilizarlas en funciones. La sintaxis es exactamente la misma.

Métodos inline

Cuando definimos un método en una clase, sólo escribimos su cabecera y lo implementamos aparte. Esto le otorga mayor organización y claridad a nuestro código, aunque a veces resulta un poco molesto tener que separar la implementación de un método cuando éste consta sólo de un par de líneas.

C++ permite definir *métodos inline,* que se declaran y se implementan en la misma declaración. La ventaja que tienen, además de evitarnos la separación de la declaración y la implementación, es que el compilador permite ejecutarlos más rápido que un método normal, mediante la reserva de memoria con anticipación.

Sin embargo, debemos tener cuidado a la hora de definir un método como inline, ya que su virtud puede jugarnos en contra si no sabemos elegir correctamente. Si el

método es corto, optimizamos el uso de memoria y la ejecución resulta más rápida que si el método tuviese una definición normal. En cambio, si el método es largo o tiene muchos parámetros, la reserva de memoria termina siendo un problema y el método resulta más lento que en una definición normal.

Para definir un método como inline basta con agregar la palabra reservada *inline* antes de la declaración del mismo. Luego, el método debe cumplir con algunas restricciones:

- No puede contener variables de tipo *static.*
- No puede contener una sentencia de bucle.
- No puede contener un *switch.*
- No puede contener la sentencia *goto.*

Si el compilador no puede resolver el método como inline, lo ejecuta como normal.

Si el método es suficientemente corto, puede omitirse agregar la palabra reservada inline y el compilador lo interpretará de esta forma.

Lo siguiente es un ejemplo de método inline:

```
#include<iostream>
using namespace std;

class A {

private:
    int atributo;
public:

    // constructor inline sin la palabra reservada
    A(){ atributo =1; }
    // metodo inline con la palabra reservada
    inline int ver_atributo(){ return atributo;}

};

    int main (int argc, char *argv[]) {

        A claseA;
        cout<< claseA.ver_atributo();
        return 0;
    }
```

Script 16

Al igual que las demás herramientas que hemos presentado, la declaración de métodos inline también se aplica a funciones. Las restricciones son las mismas, sólo que la declaración inline debe ser explícita. Caso contrario, el compilador interpretará la función como normal.

Métodos friend

Sabemos que al declarar un atributo o método como privado, no podemos acceder al mismo fuera de la clase. Éste es el principio del encapsulamiento y una de las características fundamentales de la POO.

Sin embargo, C++ es un lenguaje que todo lo puede y tiene trampas para cada una de sus reglas. Precisamente, la declaración de un *método friend* es lo que nos permite romper con esa regla de encapsulamiento.

Veamos cómo funciona con un ejemplo:

Retomemos la clase elfo y comarca. La primera tiene un atributo privado *vida* y un método *set_vida,* también privado. Este método sólo puede ser accedido por otros métodos que resuelven las acciones del elfo, como el combate, la magia de curación, etc. Obviamente, si nosotros intentamos acceder al método set_vida o al atributo vida, no lo vamos a lograr porque dichos elementos son privados.

Sin embargo, en la comarca también hay enfermerías que le permiten a los elfos curarse completamente. ¿Cómo podemos hacer para tener un método en la clase *enfermeria,* que permita modificar el atributo *vida* o acceder correctamente al método set_vida de la clase elfo? Permitiéndole el acceso desde elfo a enfermeria, declarando al método que cura en la clase enfermeria como fiend.

Resumamos:

elfo:

- Atributo *vida* privado.
- Método *set_vida* privado.

enfermeria:

- Método *restaurar_vida_elfo* público. Este mismo método es el que accede al método *set_vida* de elfo y es el que es declarado como *friend*.

Para declarar como fiend a un método basta con hacerlo en la clase que dará los permisos, anteponiendo la palabra reservada friend. La declaración del mismo se hace en la clase original sin ningún agregado especial.

Veamos cómo queda nuestro código:

```
#include<iostream>
using namespace std;

// declaracion foward, necesaria para que enfermeria
// reconozca la existencia de la clase
class elfo;


/*** Clase enfermeria ***/

class enfermeria {

public:
    // metodo publico que restaura la vida del elfo
    // toma como parametro un elfo pasado por referencia
    void restaurar_vida_elfo(elfo &E);
};


/*** Clase elfo ***/

class elfo {

private:

    int vida;
    inline void set_vida(int a){vida = a;}

public:

        inline elfo(){set_vida(100);}

    inline int ver_vida(){return vida;}

    inline void recibir_danio(int d){set_vida(ver_vida()-d);}

    // se declara el metodo como friend, la cabecera es la misma
    // que la de la clase origial que lo contiene
```

```
    inline friend void enfermeria::restaurar_vida_elfo(elfo &E);

};

// En la implementacion del metodo restaurar_vida_elfo podemos acceder
//libremente a todos los metodos y atributos privados de la clase elfo

void enfermeria::restaurar_vida_elfo(elfo &E){E.set_vida(100);}


int main (int argc, char *argv[]) {


    elfo Ninfa;
    enfermeria El_Druida;

    Ninfa.recibir_danio(50);

    cout<< Ninfa.ver_vida()<<endl;

    // Ninfa.set_vida(120); es privado, no puedo acceder

    El_Druida.restaurar_vida_elfo(Ninfa);

    cout<<Ninfa.ver_vida()<<endl;

    return 0;
}
```

La declaración fordware es necesaria, ya que de otro modo el compilador no reconocería la clase elfo con anticipación. Esto es una desventaja de la herramienta porque si tenemos una división de archivos, nos obliga a insertar una declaración ajena en nuestra clase.

Por otro lado, el método friend tiene libre control sobre los atributos privados de la clase elfo. Esto significa que podría modificar directamente el atributo vida, sin pasar por el método set_vida. Si bien el objeto nacido de elfo (Ninfa) no tiene acceso a los métodos privados, dentro del método friend, el objeto pasado por parámetro tiene todo tipo de libertad. Es como si el apuntador de la clase estuviera en acceso local.

A pesar de que es el caso más interesante, el operador friend no se limita sólo a modificar el acceso a métodos, sino que también brinda la posibilidad de modificar los permisos de una clase entera. Esto se logra definiendo como amiga a la totalidad de la clase deseada En nuestro caso, dentro de elfo deberíamos definir como amiga a la clase enfermeria entera.

La sintaxis es la misma, sólo que a la hora brindar las amistades debemos hacerlo a la clase completa. Ahora, todos los elementos de enfermeria tienen pleno acceso dentro de elfo.

Veamos cómo queda nuestro código, definiendo como amiga la clase enfermeria:

```
#include<iostream>
using namespace std;

// declaracion foward, necesaria para que enfermeria
// reconozca la existencia de la clase
class elfo;


/*** Clase enfermeria ***/

class enfermeria {
```

```
public:
    // metodo publico que restaura la vida del elfo
    // toma como parametro un elfo pasado por referencia
    void restaurar_vida_elfo(elfo &E);

    // Metodo que muestra la cantidad de vida del elfo pasado
    // por parametro
    void ver_vida_elfo(elfo E);
};


/*** Clase elfo ***/

class elfo {

private:

    int vida;
    inline void set_vida(int a){vida = a;}

public:

        inline elfo(){set_vida(100);}

    inline int ver_vida(){return vida;}

    inline void recibir_danio(int d){set_vida(ver_vida()-d);}

    // se declara la clase enfermeria entera como friend
    friend class enfermeria;

};

// En la implementacion de enfermeria podemos acceder libremente a
// todos los metodos y atributos privados de la clase elfo

void enfermeria::restaurar_vida_elfo(elfo &E){E.set_vida(100);}
void enfermeria::ver_vida_elfo(elfo E){
    cout<< "vida de Elfo es: "<<E.vida;
}

int main (int argc, char *argv[]) {


    elfo Ninfa;
    enfermeria El_Druida;

    Ninfa.recibir_danio(50);

    cout<< Ninfa.ver_vida()<<endl;

    // Ninfa.set_vida(120); es privado, no puedo acceder

    El_Druida.restaurar_vida_elfo(Ninfa);

    cout<<Ninfa.ver_vida()<<endl;

    El_Druida.ver_vida_elfo(Ninfa);
    return 0;
}
```

El operador friend se puede aplicar a funciones globales. La sintaxis es idéntica a la que usamos para los métodos. Luego, dentro de dichas funciones, tendremos libre acceso a los elementos privados de la clase que nos da los permisos.

La pregunta que podríamos hacernos es: ¿Esta herramienta afecta la seguridad de una clase?

Al respecto, cabe aclarar que sólo una clase puede decidir quiénes son sus funciones, métodos o clases amigas y, por otro lado, ninguna función, método o clase puede autodeclararse como amiga de otra clase.

A pesar de estas pequeñas reglas que nos respaldan a la hora de utilizar el operador friend, debemos tener bien el claro qué efectos causa el operador. No podemos olvidar, por ejemplo, que una función, método o clase amiga tiene libre acceso a todos los elementos privados de la clase que le otorgó la amistad, es prácticamente un usuario con permiso de administrador dentro de nuestros cajones.

Bibliografía

Dale, Nell y Weems, Chip. *Programación y resolución de problemas con C++,* 4° ed. ISBN-13: 978-970-10-6110-7, ISBN-10: 970-10-6110-1.

García de Jalón, Javier y otros. *Aprenda C++ como si estuviera en primero.* Escuela Superior de Ingenieros de San Sebastián (Universidad de Navarra), San Sebastián, 1998.

Gil Espert, Lluís y Sánchez Romero, Montserrat. *El C++ por la práctica. Introducción al lenguaje y su filosofía.* ISBN: 84-8301-338-X.

Pozo Coronado, Salvador. "Curso de c++" [en línea] [C++ con Clase] http://c.conclase.net/

Ruiz, Diego. C++ programación orientada a objetos. ISBN: 987-526-216-1.