

PROGRAMACIÓN EN C++

impartido por: Francisco Ortín Soler.

redactado por: Miguel Herrero Obeso (<http://www.miguelherrero.tk>)

actualizado el: 22/02/2005

agradecimientos a: Alejandro Rodriguez Gonzalez

Conceptos generales

encapsulacion: unión de código y datos en un mismo bloque (abstracción).

excepciones: mecanismo de tratamiento de errores.

asertos: condiciones que se tienen cumplir.

sobrecarga: dar distintas implementaciones al mismo identificador.

sobrecarga de operadores: modificar los operadores del sistema (+, -, *, /).

herencia: relación de generalización de clases. En C++ hay herencia múltiple.

polimorfismo: permite ejecutar los mismos métodos en distintos objetos en tiempo de ejecución.

genericidad: permite hacer funciones y clases genéricas respecto al tipo.

biblioteca estándar: estructuras de datos genéricas.

ejemplo:

```
#include <iostream>
#include <string>
using namespace std; //uso de la librería estándar

int main(){

    cout<<"introduzca su nombre: ";
    string nombre;
    cin>>nombre;
    cout<<"Hola, "<<nombre<<"."<<endl;
    return 0;

}
```

C++ ANSI / ISO:

En el include no se pone ".h" pues estamos llamando a clases. Con iostream podemos utilizar flujos de entrada /salida de forma similar a Java.

La tercera línea nos permite utilizar los elementos de la librería estándar. Sería equivalente al import de Java.

Para enlazar los archivos se necesita un solo método main de entre todos los archivos. Los parámetros del main es igual que en C. Se devuelve 0 si todo ha ido correcto y si es distinto de 0 devuelve el código del error.

El estándar ANSI / ISO permite no poner return 0 al final del main y el compilador lo añade automáticamente.

cout: console output. Es un flujo de salida a consola. El operador << está sobrecargado y en vez de mover bits a la izquierda, se usa para escribir flujos. En el flujo cout se puede escribir cualquier tipo simple (datos simples, punteros) y los objetos de tipo string. Cuando se escribe en flujos de este tipo se escribe en modo texto. Para instanciar un objeto se escribe 'clase nombreDeClase', no hace falta instanciarla con new.

En la siguiente línea se usa el flujo de entrada cin para leer el nombre. No es lo mismo char * que string. El char * es un puntero y es considerado de tipo simple. Al escribir cout<<"___" se evalúa la expresión y devuelve cout siempre lo que hace que se concatene con el siguiente << y así con el resto, por lo que se pueden concatenar expresiones con cout y varios <<. Lo mismo pasa con cin, lo cual lo habilita para concatenar entradas.

endl es un puntero a una función llamada manipulador. Nosotros vamos a utilizar tres:

endl, flush y ends.

El hacer cout no garantiza que se escriba inmediatamente (aunque garantiza que se escribirá antes de que termine el programa). Si queremos que el buffer vacíe la información al flujo de salida se usa "flush". "endl" concatena un salto de línea "\n" y un flush.

El main al destruir el objeto cout hace un flush (por eso el programa imprime lo que tiene que imprimir pero no garantiza el cuando).

ejemplo: calcular el valor máximo de un array

Nota: para incluir las librerías de C en C++ se escriben c+nombreLibrería

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

#define ELEMENTOS 10
#define INTERVALO 100
#define MAXIMO(a,b) ((a>b)?(a):(b))

int main(){

    int vector[ELEMENTOS];
    for (int i=0;i<ELEMENTOS;i++)
        vector[i]=(int)((float)rand()/RAND_MAX*INTERVALO/2);
    int maximo = vector[0];
    for (int i=1;i<ELEMENTOS;i++)
        maximo=MAXIMO(maximo,vector[i]);
    cout<<maximo;

}
```

operador terciario: es una especie de if-else en una sola expresión.

```
if (a>b)
    return a;
else
    return b;
```

es equivalente a

```
(a>b)?a:b
```

Función rand() devuelve int entre [0,RAND_MAX]. Rand_max es una macro definida en cstdlib. Si haces rand()/(float)RAND_MAX devuelve un valor entre [0,1).

Declaración de constantes

```
const unsigned int elementos = 10;
const int intervalo = 100;
```

Las macros son mucho más rápidas en tiempo de ejecución que las constantes / funciones equivalentes.

Funciones en línea

definición:

```
inline int maximo(int a, int b) {return a>b?a:b;}
```

En una función inline el compilador trata de poner el cuerpo de la función en el código como si fuera una macro, aunque solo lo hace cuando puede.

Se debe usarlo en funciones pequeñas, que se invoquen muchas veces y que no sean recursivas.

El problema es que las macros no distinguen los parámetros de entrada. Esto se puede arreglar con genericidad.

Genericidad

ejemplo: función que halla el máximo de dos cosas de cualquier tipo.

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

template <typename T>
T maximo(T a, Tb){ return a>v?a:b;}

template <typename T>
T nuevo(){return (T*)malloc(sizeof(T));}

int main(){

    int a, b;
    cin>>a>>b;
    cout<<maximo(a,b);
    double d1,d2;
    cin>>d1>>d2;
    cout<<maximo(d1,d2);

    int *p = nuevo <int>();
    //el compilador no sabe el tipo que debe devolver nuevo así que se indica
    //explícitamente
    *p=5;
    free(p);

}
```

Programación orientada a objetos

private: accesible desde la clase.

public: accesible desde todas las clases.

protected: accesible desde la clase y sus derivadas.

Cronometro
parado: int comienzo: clock_t final: clock_t
+ iniciar() + detener() getMiliSegundos():long estaParado():int

Nota: en C++ hay tipo bool con true y false, pero no es más que un entero que actúa como un booleano.

Por defecto, C++ declara los miembros como privados.

La declaración de las clases va en un ".h" y la implementación en un ".cpp".

```
#include <ctime>

class Cronometro{

    public:
        void iniciar();
        void detener();
        long getMilisegundos();
        int estaParado();
    private:
        int parado;
        clock_t final, comienzo;

}; //no olvidarse del punto y coma
```

Implementación de la clase cronómetro

```
//cronometro.cpp
#include "Cronometro.h"
#include <ctime>

int Cronometro::estaParado() { return parado};
void Cronometro::iniciar() {
    inicio=clock();
    parado=0;
}

void Cronometro::detener() {
    if (estaParado()) throw "No puedo detener si el cronometro esta parado";
    final=clock();
    parado=1;
}

long Cronometro::getMilisegundos() {
    if (estaParado()) return ((final-inicio)*1000.0)/CLOCKS_PER_SEC;
    return ((clock()-inicio)*1000.0)/CLOCKS_PER_SEC;
}
```

en C++ para decir al método la clase a la que pertenece se usa el operador :: .

Nota: precondition es una condición que debe cumplirse para que el método funcione de forma correcta.

En C++ se lanza cualquier cosa con throw. En C++ se pueden lanzar clases, variables, etc, por eso se omite el throws (aunque existe).

CLOCKS_PER_SEC es una macro de tipo long con el número de procesos por segundo y que está en ctime.

```
//prueba.cpp
#include "Cronometro.h"
#include <iostream>
using namespace std;

int main() {
    Cronometro c;
    c.iniciar();
    unsigned edad; cin>>edad;
    c.detener();
    c.getMilisegundos();
    c.detener();//aquí se lanza una excepción que no se recoge
    return 0;
}
```

Cuando en C++ no se atrapa la excepción se llama a una función de la librería estándar llamada terminate que a su vez llama a una función de la stdlib llamada abort(), que finaliza de un modo brusco la ejecución de la aplicación.

Al inicializar la clase, ¿qué valen sus atributos?, pues basura. Para evitar esto se emplean los constructores.

Nota: Para evitar que un archivo se incluya dos veces se debe escribir este trozo de código:

```
#ifndef Cronometro_h
#define Cronometro_h
(codigo)
...
...
#endif
```

Hay algunos trozos de código como el siguiente:

```
while(!crono.estaParado())
    cout<<crono.getMilisegundos()<<"\t"<<flush
```

que consumen muchos recursos para lo pequeños que son así que vamos a declararlos "en línea".

```
//cronometro.h

#ifndef Cronometro_h
#define Cronometro_h

class Cronometro {

    /*
    se pueden implementar los métodos en línea en la declaración de la clase
    */
public:
    int estaParado() {return parado;}
    //se puede poner inline pero no hace falta
    inline long getMilisegundos();
    //si pones inline debes definirlo en el .h
}; //fin de clase
```

```

inline long Cronometro::getMilisegundos(){
    ...
}
#endif

```

Por cuestiones de claridad se prefiere la primera definición. En Visual Studio, en la misma máquina, con la versión inline, a tiempos iguales, resultó en 15 segundos y sin inline 79.

Constructores y destructores

En C++ se pueden definir métodos para controlar la forma en que se construyen/destruyen las instancias.

Se utilizan para inicializar los objetos y que cumplan los invariantes de clase y también para gestionar los recursos del sistema.

Constructor:

- Ha de llamarse como su clase
- No devuelve nada (ni siquiera void)
- Puede recibir cualquier tipo y número de argumentos
- Si no se implementa hay uno por defecto sin parámetros

Destructor:

- Se ejecuta automáticamente al eliminar el objeto.
- No devuelve nada (ni void)
- no tiene parámetros
- Existe uno por omisión que libera el estado del objeto

```

//Cronometro.h

class Cronometro {
public: Cronometro(int);
    ...
};

//Cronometro.cpp

Cronometro::Cronometro(int iniciado){
    if (iniciado) iniciar();
    else{
        parado=1;
        inicio=final=0;
    }
}

```

Ejemplo:

```

#include <iostream>
#include "Cronometro.h"
using namespace std;

int main(){

    Cronometro c1(1), c2(false);
    cout<<c2.getMilisegundos()<<endl;
    cout<<c1.estaParado()?"parado":"iniciado"<<endl;
    Cronometro c3;//esto da error porque una vez que se crea un
    constructor
    //el constructor por defecto desaparece
}

```

```
}
```

Nota: cuando creamos un constructor, el constructor por defecto se elimina (el compilador ya no lo tiene en cuenta).

Ejemplo: Gestión de Memoria

Hecho en C:

```
#include <stdio.h>
#include <stdlib.h>

int global=3; //las variables globales se guardan al principio de la
memoria

int *stack(int a){
    int temp = a;
    return &temp;
}

int *heap(int b){
    int *temp=(int *)malloc(sizeof(int));
    *temp=b;
    return temp;
}

int main(){
    int local = 4;
    int *p=stack(local); //lo normal es que dé error en ejecución
    printf("%d\n", *p);
    p=heap(local); printf("%d", *p); //devuelve un puntero a 4
    free(p);
}
```

el mismo programa en C++: uso de **new** y **delete**:

T* new T: crea una zona de memoria (heap) para que quepa T y devuelve un puntero a esa zona.

```
int *p = new int;
```

new es un operador, no una función.

Para usar el constructor se usa así:

```
Cronómetro *p = new Cronómetro(c);
```

Si el constructor no tiene parámetros, podemos omitir los paréntesis.

Esto es muy útil si usamos el operador []:

```
char *cadena = new char[80];
```

Al hacer el **new** se puede utilizar en vez de 80 una variable pues este array se crea en la memoria dinámica en tiempo de ejecución y su tamaño puede ser variable;

delete T*: elimina la zona de memoria creada con **new**.

```
delete ptr;
```

si la clase tiene destructor se llama ahora.

Hay otra sintaxis utilizando corchetes:

```
delete[] cadena;
```

se emplea cuando se haya creado con new un array en tiempo de ejecución. Entre los corchetes NO se pone el tamaño.

Al ser operadores (new y delete) se pueden sobrecargar.

Si no hay memoria para el new, se lanza la excepción bad_alloc de la stdlib en el header <new>. Si el puntero apunta a 0 (NULL) el delete no hace nada.

```
int n; cin>>n;
Cronometro *vector = new Cronometro [n];
```

¿y si en la línea anterior se quiere usar un constructor con parámetros?

No se puede. No puedes invocar un constructor con parámetros en una creación de un array dinámico.

ejercicio: hacer una clase Vector que permita gestionar un número dinámico fijo de enteros

```
//Vector.h
class Vector {
    unsigned elemento;
    int *memoria;
public Vector(unsigned tam) { memoria = new int[elemento=tam];}
~Vector() { delete[] memoria;}
    unsigned numeroElementos() { return elemento;}
    void getElemento (unsigned);
    int setElemento(unsigned);
};

//Vector.cpp
int Vector::getElemento(unsigned i) {
    if (i>=elementos) throw "Fuera de rango";
    return memoria[i];
}

void Vector::setElemento(unsigned i,int num)
    if (i>=elementos) throw "Fuera de rango";
    memoria[i]=num;
}

//El main sería así

#include "Vector.h"
#include <iostream>
using namespace std;

int main() {

    int n; cin>>n;
    Vector tabla(10);
    for (int i=0; i<tabla.numeroElementos();i++);
        tabla.setElemento(i,(i+1)*n);
    for (int i=0; i<tabla.numeroElementos(); i++)
        cout<<n<<'x'<<(i+1)<<'='<<tabla.getElemento(i)<<endl;
```

```

    /*
    rellenado de un vector bidimensional, un array de punteros a vectores
    para poder inicializar cada vector a un valor, pues declararlos como un
    solo array no permite usar constructores con parámetros
    */

    Vector *vector[10];
    //si lo hacemos en la heap: Vector **v=new(Vector *)[m]
    for (int i=0; i<10;i++)
        vector[i]=new Vector(10);
    for (int i=0;i<10;i++)
        for (int j=0;j<10;j++)
            vector[i]->setElemento(j, (j+1)*(i+1));

    //borramos la estructura
    for (int i=0;i<10;i++) delete vector[i];
}

```

Clases amigas

Los miembros de una clase A declarada como amiga de otra clase B puede acceder a todos los miembros de las instancias de B (saltándose la encapsulación). El ejemplo más frecuente es la clase Nodo, hecha puramente para ser tratada por una clase Lista. Declarando Lista como amiga de Nodo permitiría a esta crear Nodos a su antojo. Podríamos crear todos los métodos y atributos privados para que solo Lista pueda acceder a ellos.

También hay funciones amigas, que acceden a los miembros de su clase amiga. Hablamos de funciones, no tienen por qué ser métodos de una clase. Para declarar a una clase como amiga escribimos:

```
friend class nombreDeClase;
```

Nota: para meter dos o más clases en un archivo, si necesitamos una clase que se declara después tenemos que escribir la cabecera de la clase (como ocurre con los métodos). Para ello se escribe `class nombreDeClase;`

Parámetros por omisión

En C++ se pueden usar valores por omisión de los parámetros. Se indican en la declaración o en la definición, **nunca** en ambas.

```

void f(int a, float b=0.0, char c='_', char *s="_"){
    cout<<a<<b<<c<<s<<endl;
}

```

Los parámetros por omisión siempre están más a la derecha, nunca a la izquierda de un parámetro "corriente".

USO:

```

f(1,1,'1',"1"); //1 1 1 1
f(1,1,'1'); //1 1 1 _
f(1,1); // 1 1 _ _
f"(1,2,"hola"); //error, intenta asignar "hola" a un char

```

ejemplo:

```

class Vector{
public:

```

```

        Vector (unsigned=10);
    };

    //uso del constructor

    Vector v1(19),v2; //nótese la diferencia entre los parámetros por omisión
    //y la sobrecarga de métodos.

```

Operador de ámbito (en detalle)

```

#include <cstdio>

class A{
    int n;
public:
    A(int n) {A::n=n}
    void printf(char *s) { ::printf(s);}
    // Como printf es una función global, a la izquierda de :: no va nada
}

```

La declaración general es `Ámbito::identificador`. Sirve para llamar a funciones o para hacer referencia a una variable determinada. En C++ también existe `this` para hacer referencia a los atributos de una clase.

Espacios de nombres

C++ permite agrupar abstracciones en espacios de nombres (namespaces). Cualquier cosa puede definirse ahí: clases, macros, variables...

Se usa para:

- evitar la colisión de nombres
- agrupar abstracciones relacionadas entre sí
- subdividir el problema

ejemplo:

```

//coleccion.es.h
namespace Colecciones { //declaración de clases
    class Vector {...};
    class Lista {...};
    class Cola {...};
}

//coleccion.es.cpp
namespace Colecciones{
    unsigned Vector::numeroElementos() { return elementos; }
    ...
}

//en el fichero principal

#include "Colecciones.h"

int main(){
    Colecciones::Vector v(19);
    using Colecciones::Lista;
    Lista lista;
    using namespace Colecciones;
    Cola c;
}

```

Uso de asertos

Precondición: condición que se ha de cumplir en la ejecución de un método para que esta pueda realizarse de forma correcta.

Postcondición: condición que ha de cumplirse si el método se ha ejecutado de forma esperada.

Invariante (de clase): condición que ha de cumplirse para que la instancia de la clase sea íntegra.

ejemplo: para nuestra clase cronómetro:

```
!parado || (final>comienzo)
```

Los asertos detienen la ejecución del programa y muestran el error.

```
#include <cassert>

void assert (int condicion)
```

Si definimos NDEBUG los assert no se procesan. El mensaje de error muestra el fichero y la línea que contiene el error. Después de mostrar eso, llama al método abort().

Se pueden facilitar las cosas creando un método privado que compruebe las invariantes de clase. ¿Cuándo debemos comprobar las invariantes? se deben comprobar, al menos, antes y después de los métodos además de las pre / post condiciones. Esto solo es una ayuda al programador, para enviar errores al usuario se usan excepciones.

Referencias o alias

Una referencia es un identificador cuya zona de memoria es la variable a la que hace referencia (como tener otro nombre para la misma variable).

Las referencias se declaran así:

```
int a=1;
int &r = a;
```

siempre hay que inicializar las referencias.

No son punteros, los dos contienen y son lo mismo. 'r' no apunta a 'a', es 'a'.

Con referencias, desde que se inicializan hasta que termina su ámbito, no se pueden modificar (al contrario que los punteros).

Una referencia de tipo T se ha de inicializar con un lvalue de tipo T.

T <- lvalue T

ejemplos:

```
int &r=3; //no es un lvalue
const c=3;
int &r=c; //no se puede modificar una constante
char ch='a';
int &r=ch; //no son de igual tipo
```

Si la referencia es de tipo const, la puedo inicializar con cualquier T o valor promocionable a T.

ejemplo:

```
const double &rd=3;//3 es entero que promociona a double
```

Objetos constantes

```
#include <iostream>
using std::out;
class Entero{
public:
    int n;
    Entero ( int n) { Entero::n=n; }
    int getEntero() { return n; }
    void setEntero(int a) { n=a; }
}

//main

void main() {
    Entero e(1);
    const Entero ce;

    /*un objeto constante no permite que se modifique su estado. El estado es
    la unión de todos los valores en tiempo de ejecución */

    e.n++;
    ce.n++;//incorrecto
    ce.setEntero(-1);//incorrecto
    cout<<ce.getEntero();//incorrecto

    /*para C++ todos los métodos son, por omisión, modificadores. Hay que
    indicar explícitamente los métodos selectores, es decir, los que no
    modifican el estado del objeto */
```

Hay que declarar el método así:

```
int getEntero()const { return n}
```

Se pone const en la declaración y en la definición. Ahora ce.getEntero(); funciona. Si ponemos const en un método modificador, el compilador da un error.

Paso de parámetros (en profundidad)

En C++ hay tres pasos de parámetros: valor, dirección y referencia.

por valor: el parámetro formal es una copia del parámetro real. Al devolver un parámetro en un método crea también otro objeto para devolverlo.

por dirección: el parámetro es la copia de un puntero.

por referencia: el propio elemento es el parámetro, pero se le trata a través de un alias.

ejemplo:

```
#include "entero.h"

Entero global(1); //una variable
```

```

Entero valor(Entero e) { e.setEntero(-1); return global; }
Entero direccion (Entero *e) { e->setEntero(-1); return &global; }
Entero &ref(Entero &e){ e.setEntero(-1); return global;}

int main(){
    Entero local(2);//otra variable;
    valor(local).setEntero(-2);
//crea dos copias, una que va a la función como parámetro y otro que se
//devuelve
    direcc(&local)//crea dos copias del puntero.
    ref(local).setEntero(-2);

    std::cout<<global.getEntero()<<local.getEntero();

```

ejemplo:

```

#include "Vector.h"
#include <iostream>
using namespace std;

void muestra (Vector v){
    for (int i=0;i<v.numeroElementos();i++)
        cout << v.getElemento(i)<<' \';
    cout<<endl;
}

int main(){
    Vector w(10);
    for (int j=0; j<w.numeroElementos(); j++)
        w.setElemento(j,j);
    muestra(w);
}

```

w almacena numElementos y un puntero a memoria que apunta a un array de n elementos en la heap. Al pasarlo por valor, se copia el estado pero apunta al mismo trozo de memoria que el anterior.

La v (local) al salir de la función se borra y su destructor elimina el array de elementos, por lo que el vector original w tiene un puntero a basura. Para solucionar esto, nosotros mismos implementaremos la forma en que se duplican los objetos.

Objetos no simples: aquellos que C++ no los representa en su totalidad (como en el ejemplo anterior). Un objeto no es simple si tiene un atributo de tipo puntero o tiene un atributo de tipo no simple.

Para implementar la duplicación de objetos, C++ ofrece un mecanismo llamado el **constructor de copia**. Todos los objetos tienen un constructor de copia por omisión que realiza una copia binaria del estado del objeto.

Sintaxis de ejemplo:

```

Vector (Vector &v)
Vector (const Vector &v)

```

Normalmente se usa la segunda versión pues no es normal modificar el objeto a clonar. Vamos a modificar la clase vector para que acepte lo anterior:

```

//Vector.h
class Vector {
    ...

```

```

    public:
        Vector (const Vector &);
        ...
};

//Vector.cpp
Vector::Vector(const Vector &vector){
    memoria = new int[elementos = vector.elementos];
    for (int i=0;i<elementos;i++)
        memoria[i] = vector.memoria[i];
}

```

C++ se encargará de llamar a esta función cada vez que se desee hacer una copia del objeto (como en los pasos por valor). Existe otra solución para abordar el problema anterior. Nosotros hemos permitido las copias implementando un método para ello. Pero en métodos complejos en lo que no se deba hacer una copia podemos prohibirla.

Para prohibir la copia de un objeto simplemente hemos de declarar el constructor de copia en la parte privada de la clase.

ejemplo:

```

class Vector {
    ...
private:
    Vector (const Vector &){}
    ...
};

```

Reglas sobre el paso de parámetros:

Tipos simples:

¿Se modifica el parámetro real? >>> referencia o dirección
 ¿No se modifica el parámetro real? >>> valor

Objetos:

¿No se debe modificar el parámetro formal? referencia constante
 ¿Modifica el parámetro real? referencia o dirección
 ¿Modifica el parámetro formal pero no el real? valor

Devolución por referencia: la devolución en C++ puede ser un L-value.

ejemplo: resumir getElemento y setElemento en un método.

```

int &Vector::elemento(unsigned i) {
    return memoria[i];
}
//llamada

v.elemento(i) = valor;
valor = v.elemento (i);

```

Nota: nunca devolver la dirección de una variable local.

Signatura de métodos y funciones

- ID
- número de parámetros
- tipo de parámetros

-valor devuelto

en C++ se pueden sobrecargar funciones y métodos variando el número o tipo de los parámetros.

ejemplo:

```
int menos(int a){return -a};
double menos(double a){return -a};
int menos(int a,int b){return a-b};
double menos(double a,double b){return a-b};
long int menos(int a,int b){return a-b}
/*La ultima linea no compila porque solo se varía el tipo devuelto
respecto de la tercera linea */
```

Se puede sobrecargar un método modificador con otro selector de idéntica signatura.

```
class A{
public:
void m() const { cout<<"Selector"; }
void m() { cout << "Modificador"; }

void F(const A &a) { a.m();}

int main(){
A c;
const A ca;
a.m(); //modificador
ca.m(); //selector
F(a); //selector
}
```

Sobrecarga de operadores

En C++ se pueden modificar la semántica de los operadores del lenguaje. Podemos sobrecargar los operadores con dos sintaxis, una de método y otra de función. Podemos sobrecargar los operadores binarios y unitarios.

	Operador Binario (+)	Operador Unario (-)
Método:	Vector::operator+(Vector)	Vector::operator-()
Función:	operator+(Vector,Vector)	operator-(Vector)

Nota: el operador terciario A ? B : C no se puede sobrecargar.

Restricciones:

Puedo sobrecargar un operador con sintaxis de método o de función, pero nunca con ambas. Los parámetros y valores devueltos han de hacerse por valor o referencia, nunca por dirección.

No podemos cambiar la precedencia de operadores.

No podemos crear nuevos operadores.

No podemos usar parámetros por omisión.

Tipos de constructores

```
Vector v1(10);
```

otra forma de escribir lo anterior es

```
Vector v2 = Vector (10);
```

usando el constructor de copia

```
Vector v3(v1);
```

otra forma de usar el constructor de copia

```
Vector v4=v1;
```

y otra forma más

```
Vector v5 = Vector (v1);
```

para los tipos simples

```
int n=3
```

otra forma de hacerlo

```
int n(3)
```

Uso del this

Todo método no estático (no de clase) permite acceder a la palabra reservada this. Esta es un puntero al objeto implícito.

```
class Entero {
    int n;
    public:
        Entero(int n) { this->n=n; }
        Entero *direccion() { return this; }
    ...
};

//main

int main(){
    Entero e(0), e2(2);
    cout<<e.direccion()<<e2.direccion;
}
```

ejemplo:

```
//Vector.cpp

#include "Vector.h"
#include <iostream>
#include <string>
using namespace std;

Vector::Vector (int v[], unsigned t){
    memoria=new int[elementos=t];
    for (int i=0; i<t; i++)
        memoria[i]=v[i];
}

Vector::Vector (const string &s){
    memoria=new int [elementos=s.length()];
    for (int i=0; i<elementos; i++)
        memoria[i]=s[i];
}
```

```

}

Vector Vector::operator+ (const Vector &v){
    Vector temp(elementos + v.elementos);
    int i;
    for (i=0; i<elementos; i++)
        temp[i]=memoria[i]; // temp[i]=(*this)[i];
    for (int j=0; j<v.elementos; j++, i++)
        temp[i]=v[j];
    return temp;
}

int Vector::operator==(const Vector &v) const{
    if (elementos!=v.elementos)
        return 0;
    for (int i=0; i<elementos;i++)
        if (memoria[i]!=v[i])
            return 0;
    return 1;
}

Vector &Vector::operator+= (const Vector &v){
    return *this=*this+v;
}

int Vector::operator[](unsigned n) const{
    if (n>=elementos) throw "Fuera de rango";
    return memoria[n];
}

int &Vector::operator[](unsigned n){
    if (n>=elemento) throw "Fuera de rango";
    return memoria[n];
}

ostream &operator<<(ostream &o, const Vector &v){
    o<<'[';
    int i=0;
    for (;i<v.numElementos()-1;i++)
        o<<v[i]<<",";
    if (v.numElementos())
        o<<v[i]<<' ';
    o<<']'<<flush;
    return o;
}

//programa principal

int main(){
    int v[]={1,2,3},n;
    cin>>n;
    Vector v1(n);
    for (int i=0; i<v1.numElementos();i++)
        v[i]=i;
    Vector v2(v,sizeof(v)/sizeof(v[0]));
    cout<<v1+v2<<endl<<v1[0]<<endl;
    cout<<"¿Iguales?"<<v1==v2<<endl;
    string s("hola");
    Vector v3(s);
    cout<<v3<<endl;
}

```

Construcción anónima: construir un objeto sin almacenarlo a una variable al ser un objeto y no un puntero, se destruye automáticamente.

```
cout<<Vector(string("hola"))<<endl;
```

Conversión de tipos mediante constructores

Un constructor con un parámetro identifica una promoción del tipo del parámetro al tipo de la clase.

```
#include "entero.h"
#include <iostream>
#include <string>
using namespace std;

void f(const Entero &e){ cout<<e.getEntero()<<endl; }
void h(const string &s) { cout<<s<<endl; }

int main(){
    int n=3;
    f(3); //el lenguaje crea un Entero y lo construye usando como
    parámetro el 3.
    h("hola");
    Entero a=4; //C++ hace a=entero(4);
}
```

Hay una palabra reservada que evita la promoción implícita, `explicit`.

```
explicit Vector::Vector(unsigned n);
```

```
#include "Vector.h"
#include <iostream>
#include <string>
using namespace std;

int main(){
    string s;
    cin>>s;
    Vector v(s);
    cout<<v+s<<endl;
    cout<<v+6<<endl;
    cout<<s+v<<endl<<3+v<<endl;
}
```

Nosotros no hemos implementado un operador `+` con strings, pero el string puede promocionar a vector y usar el operador de suma de vectores.

```
cout<<v+6<<endl
```

La línea anterior no promociona porque hemos usado `explicit` en el constructor

```
cout<<s+v<<endl<<3+v<<endl;
```

`s+v` no promociona porque el tipo conocido ha de estar a la izquierda.

```
class Vector{
public:
    Vector operator+(int n) const{
        Vector v(elementos+1);
        int i=0;
```

```

        for (;i<elementos;i++)
            v[i]=(*this)[i];
        v[i]=n;
        return v;
    }
    ...
};

Vector operator+(int n, const Vector &v){
    Vector w(v.numElementos()+1);
    w[0]=n;
    for (int i=0;i<v.numElementos();i++)
        w[i+1]=v[i];
    return w;
}

```

El siguiente programa ¿funciona? ¿Qué hace?

```

Vector w(v);
v=w;
w[0]=-7;
cout<<v<<endl;

```

Todo objeto puede asignarse a otro del mismo tipo porque C++ usa un operador de asignación por omisión. En C++ la asignación por omisión se hace realizando una copia binaria del objeto a asignar.

La primera línea usa el constructor de copia pero la asignación no duplica la memoria heap.

El problema es muy parecido al del constructor de copia. Por lo tanto si un objeto necesita un constructor de copia, necesitará el operador de asignación. Al igual que en el constructor de copia se puede prohibir poniendolo en la parte privada.

Vamos a implementar el operador de asignación a la clase Vector.

```

//Vector.h

class Vector{
    ...
public:
    Vector &operator=(const Vector &t){
        delete[] memoria;
        memoria=new int[elementos=t.elementos];
        for (int i=0; i<elementos;i++) memoria[i]=t[i];
        return *this;
    }
}

```

debemos comprobar si la variable que se pasa no es la misma pues v=v casca en ejecución. Para ello tenemos que comprobar si los objetos estan en direcciones distintas.

```

if (this!=&t) {...}

```

Este operador tiene varias particularidades, pero entre ellas cabe destacar que no se hereda.

En la librería estándar hay una clase vector muy rápida y genérica (hecha con templates)

```

#include <vector>

```

Trae funciones como resize() para cambiar el tamaño. Nosotros tenemos que implementar la salida por pantalla pues al ser una clase genérica no hay salida por pantalla específica.

También permite hacer vectores de vectores:

```
vector<vector<int>> tablas;
```

Otros métodos:

push_back() para insertar al final y crea elementos si no hay espacios.

Coche
color: string
id: unsigned
numCoches: unsigned
numRuedas: unsigned
getColor(): string
getNumCoches(): unsigned
getNumRuedas(): unsigned

Variables de clase: aquella que es inherente a la clase y no a cada una de sus instancias.

```
//Coche.h
#include <string>
using namespace std;

class Coche{
    string color;
    unsigned id;
    static unsigned numCoches,numRuedas;
    //no podemos asignar los valores estáticos aquí
public:
    Coche(const string &s) {color=s; id=++numCoches; }
    const string &getColor(){ return color; }
    static unsigned getNumCoches(){ return numCoches; }
    /* La definición de las variables de clase se hacen en el cpp*/

//Coche.cpp

#include "coche.h"

unsigned Coche::numCoches=0;
unsigned Coche::numRuedas=4;

//programa principal

int main(){
    cout<<Coche::getNumRuedas()<<endl;
    Coche ibiza("blanco");
    cout<<ibiza.getColor()<<ibiza.getNumCoches()<<endl;
    cout<<Coche::getNumCoches()<<endl;
}
```

Restricciones

- 1-En un miembro/método de clase no se puede acceder a la palabra this.
- 2-Desde un método/miembro de clase no se puede acceder a un miembro de instancia.
- 3-Desde un método de clase no se puede invocar a un método de instancia.
- 4-Un método de clase no puede ser virtual porque el polimorfismo afecta a instancias no a clases.
- 5-Un método de clase no puede ser selector.

Clases de Utilidad

Son clases cuyos miembros son estáticos.

Por ejemplo la clase java.Math.

Sirve para tener funciones de tipos que no tiene clase (tipos simples). Queremos que nadie construya objetos, así que ponemos el constructor en la parte privada.

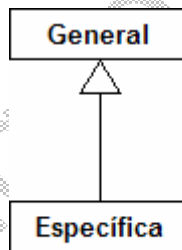
<<utility>> Matemáticas
+ PI: double + e: double
- Matemáticas() + seno(n:double):double + coseno(n:double):double + potencia(n:double):double

ejemplo:

```
cout<<Matemáticas::seno(Matemáticas.PI/2);
```

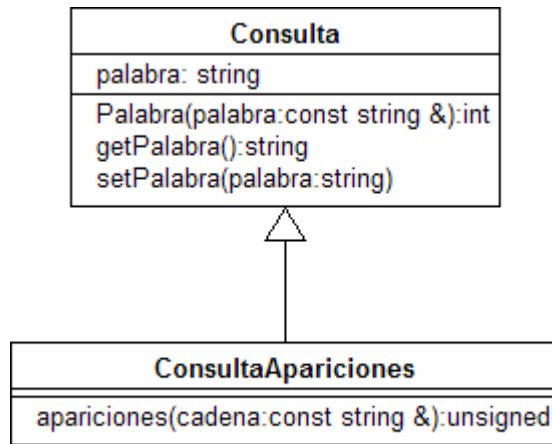
Herencia:

Es una relación de generalización entre clases.



La clase general se suele denominar clase base o superclase, la específica se suele denominar clase derivada o subclase. Es una relación transitiva. Mediante herencia, la clase derivada adopta el estado y comportamiento de la clase base. Se usa para ampliar y/o especializar.

ejemplo: herencia como mecanismo de ampliación.



```

//Consulta.h

#include <string>
using std::string;

class Consulta{
    string palabra;
    public:
        Consulta(const string &s=" ") { palabra=s; }
        const string & getPalabra() const {return palabra; }
        void setPalabra(const string &s) {palabra=s;}
        int evaluar(const string &) const;
};

class ConsultaApariciones:public Consulta{
    public:
        int apariciones(const string &)const;
};
  
```

Constructores en la herencia

- 1-Los constructores no se heredan.
- 2-Un constructor derivada ha de invocar siempre ántes de su ejecución a un constructor base. El mecanismo para invocar al constructor base se llama lista de inicialización.

ejemplo:

```

class A{
    int n;
    public:
        A(int a){ n=a; }
};

class B:public A{
    string s;
    public:
        B(int n,string a):A(n) {s=a;}
};
  
```

//la lista de inicialización es dos puntos y, entre comas ,la lista de constructores base

- 3-Por omisión la lista de inicialización esta compuesta por los constructores base sin parámetros.

```

B(string a){ s=a; }
  
```

ejemplo:

```
class Consulta{
    public:
        Consulta(const string &s){ palabra s; }
        ...
};

class ConsultaAparicion:public Consulta{
    public:
        ConsultaAparicion(const string &s): Consulta(s) {}
        ...
};
```

ejemplo:

```
class A{
    public: A(){ cout<<"ca"; }
           ~A(){ cout<<"da"; }
};

class B: public A{
    public: B(){ cout<<"cb"; }
           ~B(){ cout<<"db"; }
};

class C: public B{
    public: C(){ cout<<"cc"; }
           ~C(){ cout<<"dc"; }
};

int main(){
    C c;
}
```

Por pantalla sale: ca cb cc dc db da

Nota: en la clase string.h hay un método 'find' que devuelve la posición de una palabra en un texto. Si el texto es vacío o no se encuentra la palabra, el atributo de la clase string 'npos' es distinto de cero. El método find está sobrecargado, se puede pasar la posición en la cual empezar a buscar dentro del texto.

```
//consulta.cpp

#include "consulta.h"
#include <string>
int Consulta::evaluar (const string &texto) const {
    return texto.find(palabra)!=string::npos; }

unsigned ConsultaApariciones::apariciones (const string &texto) const {
    int pos=0;
    unsigned apariciones=0;
    while ((pos=texto.find(palabra,pos))!=string::npos){
        pos+=palabra.length(); //esto es equivalente a pos++;
        apariciones++;
    }return apariciones;
}

//programa principal

#include "consulta.h"
#include <iostream>
#include <string>
using namespace std;
```

```

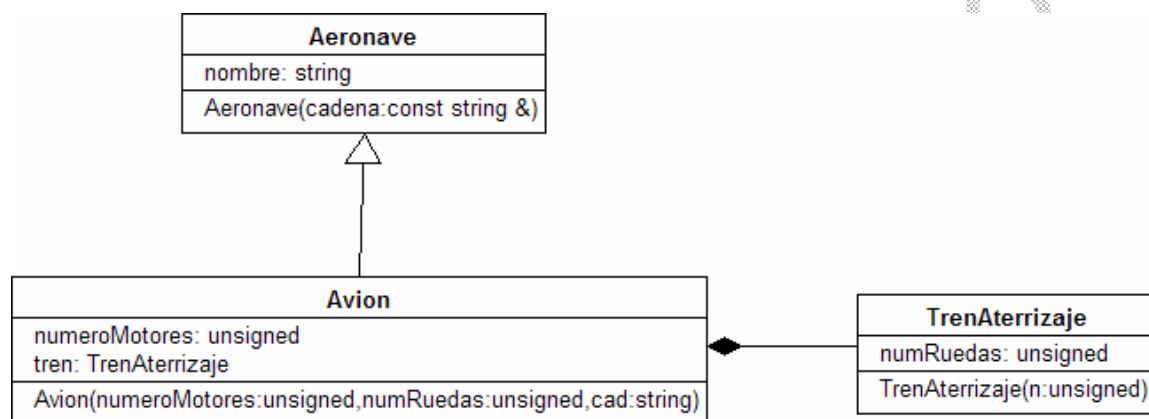
int main(){
    string frase,palabra;
    cout<<"Palabra";
    cin>>palabra;
    cout<<"Frase";
    getline(cin,frase);
    cout<<Consulta(palabra).evaluar(frase)<<endl;
    ConsultaApariciones ca;
    ca.setPalabra(palabra);
    cout<<apariciones(frase)<<endl;
}

```

Nota: cin lee palabra a palabra. Nosotros queremos una linea entera. Vamos a usar getline.

Una vez se ejecute el destructor de la clase se llama al destructor de la clase base.

Uso de la lista de inicialización para inicializar agregados:



```

class Aeronave{
    string nombre;
public:
    Aeronave (const string &s) { nombre=s; }
};

class TrenAterrizaje(unsigned nr) { numeroRuedas=nr; };

class Avion: public Aeronave{
    unsigned numeroMotores,TrenAterrizaje tren;
public: Avion(unsigned nm, unsigned nr, string n):
    Aeronave(n),tren(nr),numeroMotores(nm) {}
};

```

La composición de el TrenAterrizaje debe construirse antes de la clase. Obsérvese que llamamos al objeto tren y no a la clase TrenAterrizaje.

Ocultación de Métodos

Una clase derivada que implemente un método con igual identificador que otro de su clase base, oculta al método base, incluso aunque tenga distinta signatura.

```

class A{
public:
    void m1() const { cout<<"A::m1()"; }
    void m2() const { cout<<"A::m2()"; }
};

```

```

};

class B:public A{
public:
void m1() const{cout<<"B::m1()"; }
void m2() const{cout<<"B::m2(int)"; }
};

int main(){
B b;
b.m1();
b.m2();
}

```

b tiene cuatro constructores pero por pantalla sale:

B::m1() porque oculta a A::m1()

pero b.m2 tiene parámetros, aunque no se los pasamos ¿llamará a m2 de A?, no porque lo oculta luego produce un error.

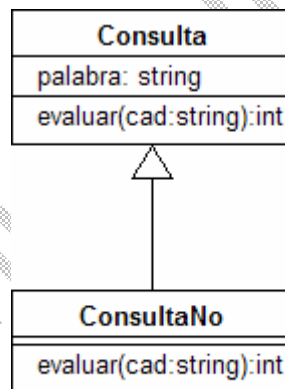
Para llamar explícitamente a los métodos se escribe lo siguiente:

```

b.m2(1);
b.A::m2();
b->A::m1();

```

ejemplo:



```

#include "consulta.h"
#include <string>

class ConsultaNo:public Consulta{
public:
int evaluar(const string &);
ConsultaNo(const string &s): Consulta(s) {}
};

int ConsultaNo::evaluar (const string &s)const{
return !Consulta::evaluar(s);
}

```

Polimorfismo

En C++, siempre que trabajemos con herencia pública, existe el polimorfismo (por defecto la herencia es privada). Para trabajar con polimorfismo hemos de trabajar con punteros o

referencias a un objeto de una clase derivada. De esta forma habilitamos la promoción de la clase derivada a la clase base.

Nota: la promoción de tipos se suele llamar cast implícito.

En C++ existe una promoción ascendente de tipos de clases en el polimorfismo. Se le suele llamar 'upcast'.

ejemplo:

```
void ref(Consulta &c) {}
void dir(Consulta *p) {}

int main(){
    Consulta c("hola");
    ConsultaNo cn("hola");
    ConsultaApariciones ca("hola");

    ref(c);
    ref(ca);
    ref(cn);
    dir(&ca);
    dir(&c);
    dir(&cn);
    Consulta *ptr;
    ptr = &ca;
    ptr = &c;
    Consulta &r1=ca,&r2=cn;
}
```

Al igual que en Java, con polimorfismo solo se puede acceder a los métodos de la clase base que hemos referenciado o apuntado.

```
r1.apariciones("hola"); //da error de compilación
```

ejemplo:

```
int main(){
    string palabra="hola",frase"hola mundo";
    Consulta *p = new ConsultaNo(palabra);
    cout<<p->evaluar(frase)<<endl;
}
```

En C++ no existe el enlace dinámico, es decir, aunque redefines un método en una clase derivada, al usar polimorfismo se llama al método de la clase base y no al de la derivada. La llamada se produce al método del tipo declarado en la variable. Esto se ha implementado así por cuestiones de eficiencia.

¿Cómo llamamos al método del objeto al que se apunta?, usando la palabra reservada virtual. Se define en la declaración del método de la superclase (no en la definición).

```
class Consulta{

    public:
    virtual int evaluar(const string &)const;

};
```

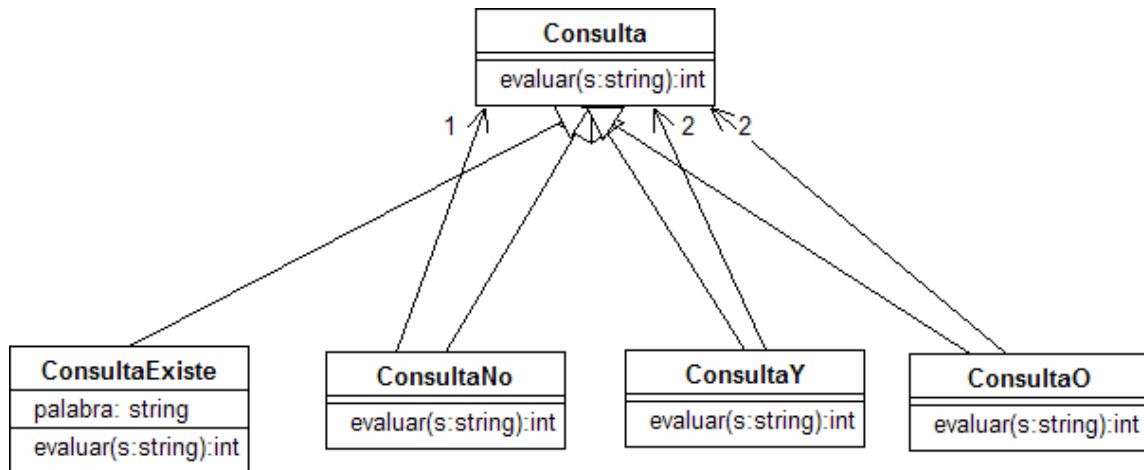
Un método derivado de otro virtual con exacta signatura también es virtual. Aunque se hereda la propiedad virtual de un método, conviene escribirla en los métodos de clases derivadas para una mayor claridad en el código.

ejemplo: diseño que nos permita hacer consultas de palabras en textos.

debemos calcular si:

- existe una palabra en un texto.
- no existe en un texto una Consulta.
- si existe un texto $C_1 \& \& C_2$, siendo C_1 y C_2 , Consultas.
- si existe en un texto $C_1 | C_2$, siendo C_1 y C_2 , Consultas.

Nota: un patrón de diseño es una solución general a un problema. Este ejemplo es un patrón de diseño llamado composición (composite).



```
class Consulta{
    public:
    virtual int evaluar(string &const {}
};

class ConsultaExiste: public Consulta {
    string palabra;
    public:
    ConsultaExiste(const string &s){ palabra=s; }
    int evaluar(const string &texto) const{
        return texto.find(palabra)!=string::npos; }
};

class ConsultaNo: public Consulta {
    Consulta *op;
    public:
    ConsultaNo(Consulta *c) { op=c;}
    virtual int evaluar(const string &texto)const{ return !op-
>evaluar(texto);}
};

class ConsultaY: public Consulta{
    Consulta *op1,*op2;
    public:
    ConsultaY(Consulta *o1,Consulta *o2): op1(o1),op2(o2){}
    virtual int evaluar(const string &texto)const{
        return op1->evaluar(texto)&&op2->evaluar(texto);
    }
};
```

```

class ConsultaO: public Consulta{
    Consulta *op1,*op2;
public:
    ConsultaO(Consulta *o1,Consulta *o2): op1(o1),op2(o2){}
    virtual int evaluar(const string &texto) const{
        return op1->evaluar(texto) || op2->evaluar(texto);
    }
};

```

Pedimos una frase al usuario y la guardamos en un string. ¿Existe la consulta Y("hola",No(O("mundo","pepe")))?

```

//programa principal

#include <stdio>
#include <string>
#include <iostream>

using namespace std;

int main(){

    cout<<"Introduce una frase: ";
    string frase;
    getline(cin,frase);

    ConsultaExiste h("hola"),m("mundo"),p("pepe");
    Consulta co(&m,&p);
    ConsultaNo *cn = new ConsultaNo(&co);
    ConsultaY cy(&h,cn);
    cout<<cy.evaluar(frase);
    delete cn;
}

```

Todo lo anterior se puede reducir a una línea.

```

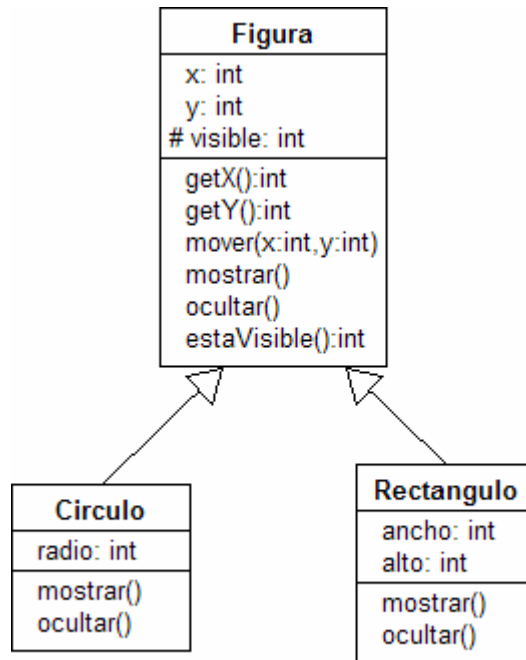
cout<<ConsultaY(&ConsultaExiste("hola"),&ConsultaNo(&ConsultaO(&ConsultaExiste("mundo"),&ConsultaExiste("pepe"))));

```

Ampliaciones de este ejercicio:

- 1) Hacer que las cardinalidades sean de 2 a n.
- 2) Añadir una nueva Consulta, ConsultaIF con tres asociaciones tal que el resultado sea op1?op2:op3
- 3) Implementar una clase ConsultaUsuario en la que su constructor, a partir de una cadena Y("hola",No(O("mundo","pepe"))) cree los objetos pertinentes y devuelve el resultado.

ejemplo: modelar un sistema de figuras.



Visible es protected, para que las clases hijas puedan acceder. Mostrar y ocultar son virtual y están vacíos (en figura). Mover oculta el objeto, lo mueve y lo muestra.

Destructores Virtuales

```

class Base{
    public:
    ~Base() {cout<<"Destructor Base";}
};

class Derivada:public Base{
    public:
    ~Derivada() {cout<<"Destructor Derivada";}
};

int main(){
    Derivada d;
    //se ejecuta primero el destructor de derivada (al revés que los
    //constructores)
    Base *p = new Derivada;
    delete p;
}
  
```

En las últimas instrucciones pueden suceder desde errores en tiempo de ejecución hasta que no se liberen bien los recursos. Al hacer delete p, borra un objeto Base, pues el puntero es de tipo Base. Para llamar al destructor de la clase Derivada hay que declararlo virtual.

```

virtual ~Base() {cout<<"Destructor Base";}
  
```

El destructor por defecto no es virtual, por lo que hay que poner un destructor virtual vacío para paliar este problema.

```
virtual ~Base() {}
```

Abstracción

Un método abstracto identifica un mensaje y no un método.

```
class Figura{  
    public:  
    virtual void mostrar() = 0; //declaración de un método virtual puro  
}
```

Nota: en C++ no se emplea el término método abstracto, si no método virtual puro.

Una clase abstracta es aquella que tiene al menos un método abstracto. No se pueden crear instancias de clases abstractas. Pero se pueden crear punteros a ellas. Tampoco se puede pasar una clase abstracta por valor.

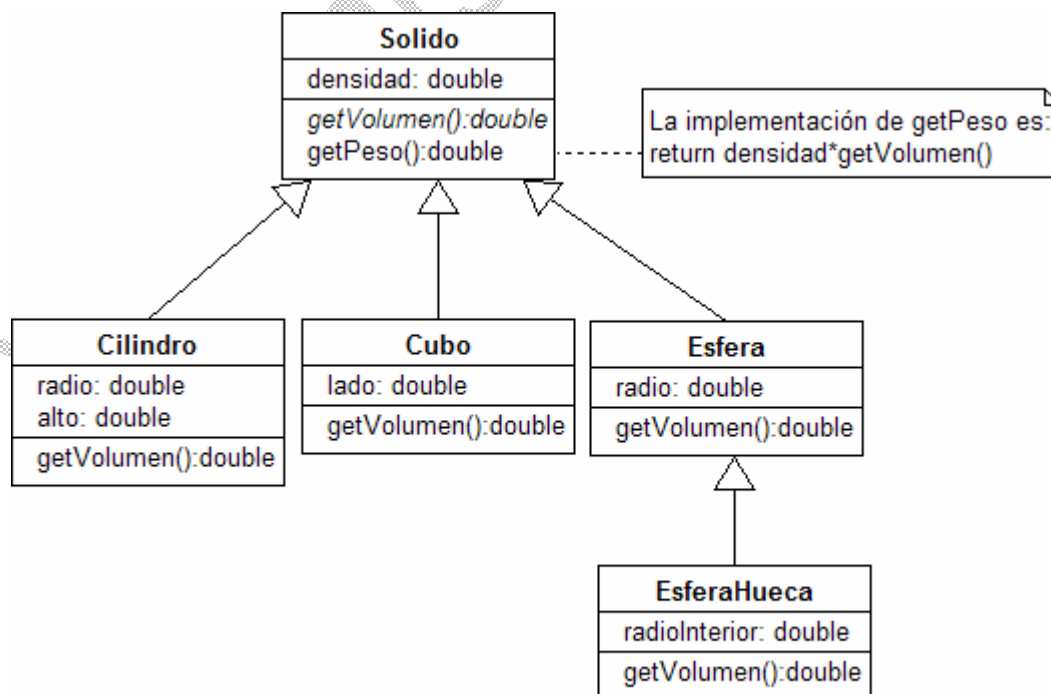
```
Consulta *p = new ConsultaExiste("h");  
Consulta &r = *p;  
Consulta c; //error de compilación.
```

En UML los métodos abstractos se indican de dos formas diferentes: en cursiva o con el estereotipo <abstract>.

Si las hojas de una clase abstracta no implementan un método abstracto, lo heredan abstracto y por lo tanto se convierten en clases abstractas. Las clases abstractas pueden tener constructores, para que las clases hijas puedan llamarlos e inicializar las variables heredadas de la clase abstracta.

Nota: las variables constantes se inicializan en la lista de inicialización.

ejemplo: vamos a usar un patrón de diseño llamado plantilla (template) y se refiere a que el método `getPeso` llama a un método abstracto (`getVolumen`) dejando parte de la implementación a éste en sus clases derivadas.

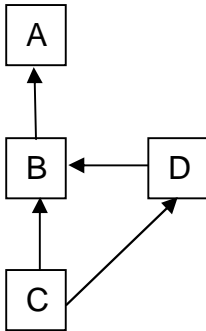


Problema: si tenemos un modelo de figuras y queremos, por ejemplo, seleccionarlas todas, ¿qué hacemos?

Tenemos que crear un vector de punteros a Figura, en la propia clase Figura, y declararlo estático. En el constructor de figuras insertamos en la lista de figuras y en el destructor las eliminamos.

Herencia múltiple

Se produce cuando un clase hereda directamente de más de una clase. Una clase solo puede heredar directamente una vez de otra clase.

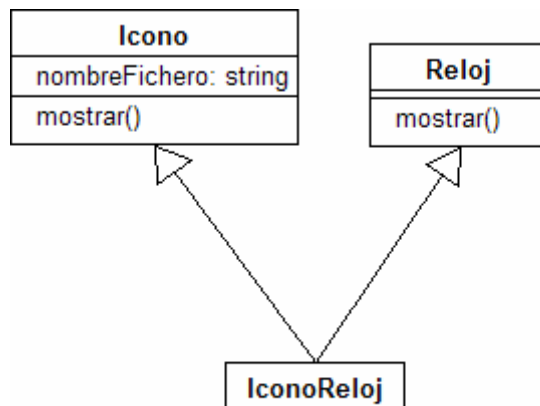


```
class A {};  
class B: public A{};  
class D: public B{};  
class C: public B, public D{};
```

Una forma de ver la herencia múltiple es la generalización: C es más específico que B y que D. herencia múltiple, al inicializar un constructor de una clase derivada, el orden de llamada a los constructores depende del orden de la declaración de la herencia, es decir, si declaramos primero la herencia de la clase Cubo y luego el de la clase Esfera, el primer constructor llamado será el de la clase Cubo.

Coincidencia de nombres

Este fenómeno sucede cuando una clase hereda un mismo miembro de dos o más clases distintas.



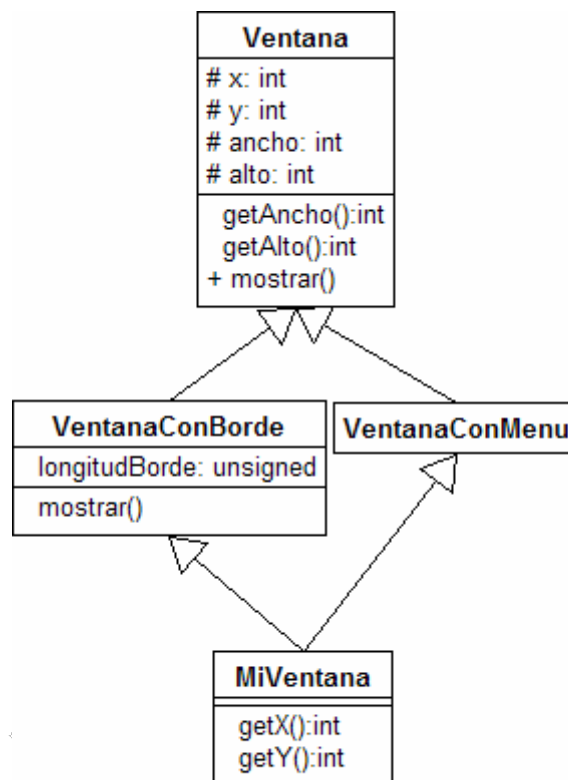
¿Qué método mostrar() se heredará? Si creamos un objeto IconoReloj y llamamos a mostrar() hay un error de compilación. El compilador no elige ninguno de los dos métodos, somos nosotros los que tenemos que especificar su uso:

```
IconoReloj ir("icono.ico");  
ir.Icono::mostrar();  
ir.Reloj::mostrar();
```

Lo más habitual es crear un tercer método mostrar() propio de la clase IconoReloj, pues ninguno de los mostrar() de las clases 'padre' saben manejar todos los datos de IconoReloj.

Herencia Repetida

Se produce cuando se hereda indirectamente más de una vez de la misma clase.

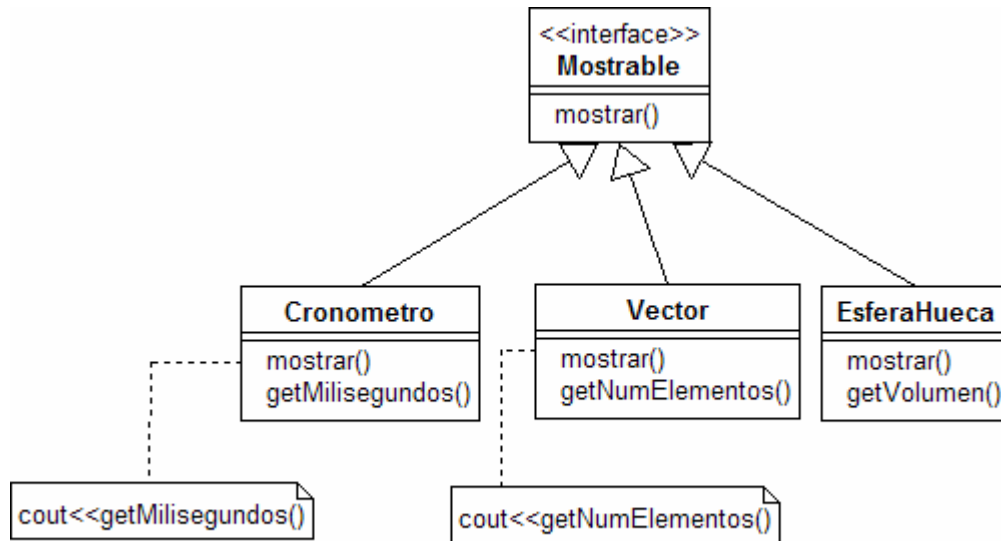


MiVentana hereda dos veces de Ventana a través de sus padres. Tenemos, dos veces, métodos y atributos de Ventana, es decir, dos x, dos y, dos getAlto, etc. La herencia se puede representar mediante un grafo dirigido acíclico.

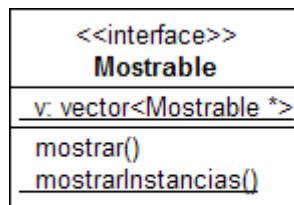
Se va a intentar no duplicar la clase, sino tener dos referencias a la misma clase. Para esto C++ utiliza la herencia virtual. Cuando comprueba que un subconjunto de un objeto ya ha sido creado no lo duplica, sino que simplemente apunta a lo creado anteriormente.

La herencia múltiple no se suele usar como especialización. Lo usual es usar la herencia simple y el resto de herencias usarlas para la implementación de funcionalidades.

ejemplo: Diseñar un programa que muestre objetos de tipo Cronometro, Vector y EsferaHueca.



Este patrón de diseño se llama puente (Bridge). Vamos a crear un vector en la clase mostrable y un método mostrarInstancias() ambos estáticos, que permitan llevar una relación de objetos Mostrables.



```

#include <vector>
using namespace std;

class Mostrable{

    public:
    static vector <Mostrable *> v;
    virtual void mostrar() const = 0;
    static void mostrarInstancias();
    virtual ~Mostrable() {}
};

class EsferaHueca: public Esfera, public Mostrable{
};
  
```

podemos hacer privado el vector y crear las instancias en el constructor y destruirlos en el destructor de Mostrable.

```

//mostrable.cpp

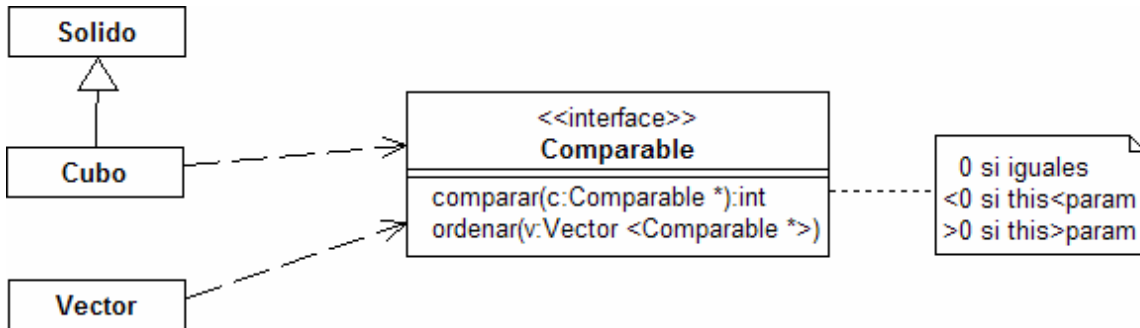
#include <vector>
#include "mostrable.h";

using namespace std;

vector<Mostrable *> Mostrable::v;
void Mostrable::mostrarInstancias(){
    for(int i = 0; i < v.size(); i++)
        v[i]->mostrar();
}
  
```

ejemplo: Queremos ordenar Vectores y Cubos en una misma estructura.

Nota: Podemos definir este tipo de relación en UML mediante una flecha discontinua.



¿Y si no queremos modificar Vector y Cubo? Creamos dos clases derivadas de estas que implementen el interfaz Comparable.

```
class Comparable {
    public:
    const virtual int comparar(Comparable *) const = 0;
    virtual ~Comparable() {}
    static void ordenar(vector <Comparable *>&);
};

class CuboComparable: public Cubo, public Comparable {
    public:
    int comparar(const Comparable * c) const {
```

Aquí nos encontramos con un problema... ¿y si nos pasan un Vector como parámetro? Además no podemos utilizar getVolumen con un Comparable. Necesitamos saber el tipo del objeto pasado por parámetro.

Podemos probar

```
((CuboComparable *) c)->getVolumen();
```

Pero esto solo funciona con CuboComparable. Con el ahormado hay que tener seguridad de que el objeto es del tipo del que esperamos.

Necesitamos una precondición que compruebe el tipo del objeto. En Java existe el operador instanceof, en C++ usamos la RTTI (run time type information), que debe ser habilitada en la mayoría de compiladores debido a su ineficiencia.

Nota: para habilitarlo en VC++ vamos a Proyecto | Propiedades | C/C++ | Lenguaje | Habilitar información de tipo en tiempo de ejecución.

Una vez activado podemos usar el cast dinámico, cuya sintaxis es:

```
dynamic_cast<T *> (puntero)
```

Devuelve 0 si el puntero no es de tipo T. En caso contrario devuelve un puntero que contiene el puntero anterior con el ahormado hecho.

Continuamos con el ejemplo que dejamos atrás...

```
int comparar(const Comparable *c) const {  
    CuboComparable *cc = dynamic_cast<CuboComparable> ( c );  
    if (!cc) throw "No puedo comparar";  
    return getVolumen() - cc->getVolumen();  
}
```

ejercicio: modificar el diseño para que un solo algoritmo de ordenación pueda ordenar con varios criterios (ascendente, descendente, primero los primos, etc.) sin modificar el mismo.

Excepciones

El manejo de excepciones permite controlar los errores y casos especiales de un modo extensible (permite ampliar su funcionalidad), reutilizable (se pueden tratar los errores de maneras diferentes) y robusto (obliga a tratar el error).

En programación estructurada hay soluciones que no aportan tantas ventajas como las excepciones, entre las que cabe destacar las siguientes:

- *retornar un valor especial*: por ejemplo, devolver cuando no se encuentra un elemento en una lista. El problema de esto es que no se amolda a todos los problemas, ¿Cómo lo usaríamos en el método pop() de una pila?
- *finalizar la ejecución*: no presenta extensibilidad ni reutilización.
- *comprobación a priori y a posteriori*: comprobar con un if, por ejemplo, una precondition o poscondición.

Con las excepciones se pretende separar el punto en el que se detecta el error del puntero en donde se maneja. Además el error debe ser controlado, por ello el mecanismo gana robustez.

Una excepción conlleva dos objetivos: transferir el flujo de la ejecución y dar información sobre el error.

En C++ las excepciones pueden ser cualquier tipo simple u objeto.

Por motivos de eficiencia, de manera estándar no se lanzan todas las excepciones que se debiera. Se realizan comprobaciones a priori y a posteriori.

Excepciones de la librería estándar

exception: la clase ancestral de la jerarquía. Nótese que no es necesario que una excepción herede de ésta pues se pueden lanzar otros objetos y tipos simples. Esta clase tiene un método const char * what() . Este método no hace nada, simplemente es una plantilla para clases derivadas. No es en abstracta, pero podría. Se mantiene no abstracta para poder usar excepciones muy simples. Está en el namespace std.

bad_alloc: es lanzada cuando se intenta reservar un trozo de memoria heap mayor del que realmente hay libre. Se encuentra en el header new.h.

invalid_argument: tiene un constructor invalid_argument(const string &) que almacena el mensaje a mostrar por el metodo what(). Se puede lanzar por ejemplo al intentar acceder a un índice fuera de un array. Se encuentra en el header except.h.

logic_error: se usa en sentencias que no tienen sentido, como hacer un `pop()` en una pila vacía. Tiene un constructor `logic_error(const string &)` que almacena el mensaje a mostrar por el método `what()`. Se encuentra en el header `except.h`.

Podemos derivar de estas clases para crear nuestras propias excepciones. Nosotros usaremos principalmente `logic_error` y `exception`, pues usaremos errores para precondiciones, poscondiciones, etc.

Nota: para convertir un formato a otro usamos streams.

```
ostreamstream buffer;  
buffer << "El rango [ "desde<< ' , ' << hasta<< " ]";  
return buffer.str();
```

Los flujos `ostreamstream` están en `sstream.h`.

Nota: para convertir un `string` a `char *` hay un método de la clase `string` llamado `c_str()`.

Lanzamiento de excepciones

Las excepciones se deben lanzar cuando se detecta un caso excepcional o error. Utilizamos la palabra reservada `throw` y con ello realizamos las tres siguientes acciones:

- Transferimos la ejecución a un manejador.
- Identificamos un tipo de excepción.
- Transferimos al manejador una determinada información.

Cuando se lanza una excepción se usa una copia de la misma. Si el objeto a lanzar no es simple, debemos implementar un constructor de copia. Si el constructor de copia es privado, no podemos lanzar la excepción.

Si lanzamos una excepción con el operador `new` (como en Java) enviamos la copia de un puntero a una excepción. Al hacer esto nos comprometemos a liberar memoria al capturarla. Este sistema es poco cómodo, por lo que se suele hacer es lanzar objetos o referencias.

En C++, por omisión, los métodos pueden lanzar cualquier cosa (al contrario de Java).

Manejo de excepciones

El manejo de excepciones se realiza con dos palabras reservadas: `try` y `catch`. Si en C++ se lanza una excepción y no se captura, se llama al método `terminate()`, que a su vez llama a `abort()` y finaliza la ejecución.

Si quiero gestionar un error uso un bloque `try`. Obligatoriamente, tras estos bloques, se han de situar uno o más bloques `catch` que determinan las excepciones a tratar. En un `match` se pueden manejar dos tipos de excepciones:

- `[const]T [&]`, se entra en el `catch` si se lanza una excepción de `const T` o derivadas.
- `[const]T *`, se entra en el `catch` si se lanza una excepción de `T` o derivadas.

Adicionalmente se puede usar un último `catch(...)` que entra cualquiera que sea la excepción lanzada.

```
int main(){  
    int desde,hasta,n;  
    cin >> desde >> hasta >> n ;  
    Vector v(desde,hasta); //posible excepción  
    for (int i=desde;i<=hasta;i++)  
        v[i]=i;
```

```

v[n]=n; //posible excepción
cout << v << endl;

try{
    Vector v(desde,hasta);
    v[n]=n;
}catch(const RangoNoValido &e){
    cerr << e.what();
}catch(const FueraRango &e){
    cerr << e.what();
    throw;
//cuando se pone throw se lanza la excepción recién capturada.

}catch(const RangoNoValido &){
//No es necesario dar nombre a las excepciones
}catch(...){
    cerr << "Error desconocido";
}
}

```

Si tenemos n sentencias en un programa y utilizamos memoria dinámica ¿Cómo la liberamos si se lanza una excepción?

```

#include <cstdio>
#include <iostream>
#include <stdexcept>

using namespace std;

void procesar(){
    //...
    throw logic_error("Error");
}

void recursos1(const char *s){
    //esta función no va a manejar el error

    FILE *f = fopen(s,"w");
    //...
    procesar();
    //...
    fclose(f);
}

int main(){
    try{
        recursos1("prueba1.txt");
    }catch(const logic_error &e){
        cerr(e.what());
    }
}

```

La función recursos1 no libera el fichero. Esto en Java se realiza cerrándolo en el finally. Ahora vamos a modificar la función para que cierre el fichero correctamente.

```

void recursos 2(const char *s){
    FILE *f = fopen(s,"w");
    try{
        //...
        procesar();
        //...
        fclose(f);
    }catch(logic_error &){

```

```

        fclose(f);
        throw;
    }
}

```

Ahora el main recibe la excepción pero el fichero ha sido previamente cerrado. Esta modificación funciona pero no deja un código claro: hace un try-catch que no maneja realmente el error. Lo que debemos hacer es utilizar destructores que liberen los recursos y usar en la medida de lo posible variables en la pila (que se destruyen automáticamente).

Los ficheros en C++ están en el header fstream y manejan este tipo de situaciones:

```

void recursos3(const char *s){
    ofstream f(s);
    //...
    procesar();
    //...
}

```

La clase ofstream se encarga de cerrar el fichero en el destructor. Esta liberación se realiza en todos los flujos.

Observemos la siguiente función

```

void f(){
    int *p = new int;
    procesar();
    delete p;
}

```

Para arreglar esta función tendríamos que hacer una clase que sobrescribiera el operador -> para devolver el puntero y no el entero. Para ello en la librería estándar hay, en el header memory, la clase auto_ptr. Esta clase elimina el objeto al que apunta aún cuando se lance una excepción.

```

#include <memory>
using namespace std;

void f(){
    auto_ptr<int> p = new int;
    procesar();
}

```

En Java se obliga a poner throws para indicar las excepciones que son lanzadas por un método. Ésta es una solución muy robusta que ayuda a documentar. En C++ no se obliga, pues se puede lanzar cualquier cosa.

Pero hay una ampliación C++ ANSI/ISO que utiliza la palabra throw y funciona de forma parecida. Se debe escribir en la declaración y definición (junto con const es la única palabra que ha de escribirse en ambos lugares). Su sintaxis es:

```

void funcion() throw (int, logic_error) {...}

```

Aquí se podrían lanzar int o logic_error y sus derivadas. Para negar el lanzamiento de excepciones se escribiría:

```

void funcion() throw {...}

```

Esto funciona en compiladores GNU, pero no en el Visual Studio .NET 2003. Éste último ignora esa parte de código. Conviene informarse sobre como actúa nuestro compilador con estas sentencias.

Si heredamos de un objeto que tiene un método que lanza tres excepciones y redefinimos el mismo en nuestra propia clase ¿Cuántas excepciones debe lanzar nuestro método redefinido?

Se deben lanzar las mismas o menos para poder mantener el polimorfismo entre ambas clases, es decir, cuanto más arriba en la jerarquía, más general debe ser la clase.

ejemplo:

```
class A{
    public:
    virtual void m1()const {...}
    virtual void m2()const throw(logic_error) {...}
    virtual void m3()const throw() {...}
};

class B: public A{
    public:
    void m1()const throw(logic_error){...}
    void m2()const throw(logic_error) {...}
    void m3()const throw(logic_error) {...}
};
```

B:m3() incumple el 'contrato' respecto a la clase padre. La clase derivada pretenda lanzar una excepción más que su padre. Esto da un error de compilación.

¿Funciona la siguiente función?

```
virtual void m3()const throw(){ throw 3;}
```

Compila pero en tiempo de ejecución se llama a una función del sistema llamada unexpected() que llama, por omisión, a terminate().

Genericidad

La genericidad es la propiedad que permite crear algoritmos y abstracciones respecto al tipo. C++ implementa esto con plantillas (templates).

Debemos declarar una plantilla en base a un tipo genérico T y realizar los algoritmos con ese tipo genérico. La sintaxis es:

```
template <typename T>
```

ejemplo:

```
template <typename T>
void intercambia (T &a, T &b){
    T aux = a;
    a = b;
    b = aux;
}
```

Si la utilizamos con objetos no simples, necesitaríamos implementar el operador = para las dos últimas líneas y el constructor de copia para la primera.

```
template <typename T>
```

```

const T & maximo(const T &a, const T &b){
    return a>b?a:b;
}

```

Devolvemos un const porque vamos a devolver uno de los dos parámetros, que son const. Con tipos no simples debemos implementar el operador > sobrecargado.

ejemplo:

```

int main(){
    int a,b;
    cout << maximo(a,b);
    Entero e1(1),e2(2);
    cout << maximo(e1,e2).getEntero();
}

```

La última sentencia no compila porque Entero no sobrecarga el operador >.

Conseguir funciones genéricas es muy difícil. Hay veces que necesitamos que se contemplen distintos fragmentos de código según el tipo que le pasemos a la función.

La genericidad es parecida al polimorfismo estático, por lo tanto los errores ocurren en tiempo de compilación y no en tiempo de ejecución (por lo tanto no se lanzan excepciones). Esto ayuda a encontrar los errores más fácilmente.

Vamos a modificar la clase Entero para que pueda utilizar la función anterior:

```

class Entero{
    ...
    int operador>(const Entero &e){
        return entero>e.entero;
    }
}

```

Ahora, la clase Entero puede utilizar la función max.

Nota: con implementar == y <, los demás operadores se pueden deducir de los anteriores, aunque es más eficiente implementarlos uno mismo.

```

cout << max(3,e1).getEntero() << endl;

```

Aquí el sistema no decide cual de los dos tipos elegir y genera un error de compilación. Debemos especificar que cree un Entero con valor 3 (por promoción de tipos) y los compare.

Otra opción es declarar dos tipos T1 y T2 pero ¿qué devuelve? Es un caso demasiado complejo.

Vamos a forzar el uso de un tipo en concreto:

```

Cout << max<Entero>(3,e1).getEntero() << endl;

```

Además de funciones, se pueden hacer clase con tipos genéricos.

Ejercicio: realizar un árbol binario de búsqueda no balanceado con tipos genéricos.

```

#include <iostream>
using namespace std;

template < typename T> class Arbol<T>;
template <typename T>
class Nodo{

```

```

T valor;
Nodo<T> *izq, *der;
unsigned elementos; //para almacenar un mismo valor varias veces

Nodo(const T &v, unsigned e=1, const Nodo<T>*i=0,const Nodo<T>*d=0){
    valor = v;
    elemento = e;
    izq = i;
    der = d;
}

friend class Arbol<T>;
void insertar(const T&);
void inorden(ostream &) const;
friend ostream & operator<< <T>(ostream &, const Arbol<T> &);

};

template <typename T>
class Arbol{

    Nodo<T> *raiz;
    static void destruir(Nodo<T> *);
    static Nodo<T> * copiar(const Nodo<T> *) ;

    public :
    Arbol<>T> { raiz = 0 ;}
    ~Arbol<>T>{ if(raiz) destruir(raiz) ;}
    Arbol<T>(const Arbol<T>f) ;
    Arbol<T> & operator=(const Arbol<T> &) ;
    void insertar(const T&) {
        if (raiz) raiz->insertar(v) ;
        else raiz = new Nodo<T>(v) ;
    }
} ;

//definimos una función fuera del ámbito de la clase
template <typename T>
ostream & operator<<(ostream &, const Arbol<T> &) ;

//Implementación

template <typename T>
void Nodo<T> ::insertar(const T &v){

    if (v==valor){ elementos++ ; return ;}
    if (valor > v) { //rama izquierda
        if (izq) izq->insertar(v) ; //si esta vacío, insertamos
        else izq = new Nodo<T>(v) ;
    }else{ //rama derecha
        if (der) der->insertar(v) ;
        else der = new Nodo<T>(v) ;
    }
}

template <typename T>
void Nodo<T> ::inorden(ostream &o)const{
    if (izq) izq->inorden(o) ;
    o << 'c' << valor << ' :' << elementos << " ) " ;
    if (der) der->inorden(o);
}

Arbol<T> ::Arbol<T>(const Arbol <T> &a){
    raiz = copiar(a.raiz) ;
}

```

```

}

template <typename T>
Arbol<T> & Arbol<T> ::operator=(const Arbol<T> &a){
    if (this != &a){ //comprobamos que no es 'a = a'
        delete raiz ;
        raiz = copia(a.raiz) ;
        return *this ;
    }
}

template <typename T>
void Arbol<T> ::destruir(Nodo<T> *n){
    if (n){
        destruir(n->izq) ;
        destruir(n->der) ;
        deleten n ;
    }
}

template <typename T>
Nodo<T> * Arbol<T> ::copiar(const Nodo<T> *a){
    if ( !a) return 0 ;
    return new Nodo<T>(a->valor,a->elementos,copia(a->izq),
        copia(a->der)) ;
}

template <typename T>
ostream & operator<<(ostream &o, const Arbol<T> &a){
    if (a.raiz) a.raiz->inorden(o) ;
    return 0 ;
}

//Programa principal

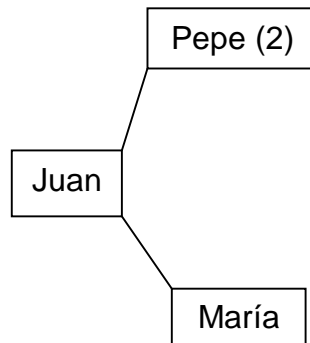
#include "arbol.h"
#include <stream>
#include <iostream>
using namespace std;
int main(){

    const char * v[] = {"pepe","juan","maria","pepe"};
    Arbol<string> arbol;
    for (int i=0; i<sizeof(v)/sizeof(char *); i++)
        arbol.insertar(v[i]);
    cout << arbol << endl;

    Arbol<int> b;
    for (int i=0; i<10; i++)
        b.insertar(i%7);
    cout << b;
}

```

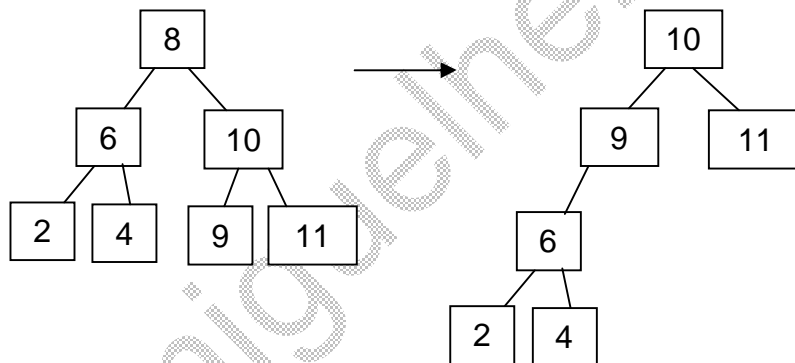
El árbol creado sería el siguiente.



Nota: cuando se crea una clase genérica, las implementaciones de los métodos tienen que ir en el .h, es decir, en la definición de la clase.

ejemplo: ampliar la estructura árbol desarrollada anteriormente con dos operaciones nuevas:

- implementar el operador [], tal que pasando un T, devuelva las veces que se repite o cero si no se encuentra ningún T en el árbol.
- borrar un nodo mediante el siguiente algoritmo:
 si hay más de una aparición, disminuimos el contador.
 si hay una aparición, borramos el nodo y lo sustituimos por un hijo.



La Librería Estándar

La STL (Standard Template Library) es un subconjunto de la librería estándar que ofrece un conjunto de colecciones y algoritmos para tratar las listas.

¿Qué parte de la librería estándar no pertenece a la STL?

- las cadenas de caracteres (string)
- la gestión de números complejos (complex)
- el límite de los tipos simples (numeric_limits)
- la gestión de flujos (stream)
- la internacionalización (locale)

Arquitectura de la STL

La arquitectura está formada por:

- contenedores, que son implementaciones de colecciones de elementos.
- iteradores, que son objetos que permiten iterar por los elementos de un contenedor.
- algoritmos, código genérico para tratar los contenedores.

Contenedores (en profundidad)

Los contenedores ofrecen implementaciones para coleccionar elementos. Hay de distintos tipos:

- **secuenciales:** sus elementos no están en función de un orden, es decir, son no ordenados.
Ejemplo: list, vector, deque (estructura en la cual se puede insertar y extraer elementos por el principio y por el final).
- **asociativos:** sus elementos siguen un orden determinado. Por defecto se ordena de forma descendente.
Ejemplo: map (a partir de una clave, obtiene un elemento), multimap (permite asociar a una clave, múltiples valores), set (conjunto, no se puede repetir elementos) y multiset (permite repetir elementos).
- **adaptadores:** son clases parametrizadas sobre otros contenedores.
stack (pila), queue (cola), priority_queue (cola de prioridad), bit_set (conjunto de bits, muy útil para su uso con máscaras), valarray (vector algebraico).

Todos los contenedores cumplen las siguientes normas:

- Todos trabajan con copias de los elementos que se insertan. Para que los elementos sobrevivan al contenedor, se suelen usar punteros a objetos.
- Por todo contenedor se puede iterar mediante el concepto de iterador.
- Las operaciones no son seguras y generalmente no se lanzan excepciones.

Requisitos que han de satisfacer los elementos a contener:

- Han de definir la posibilidad de hacer copias.
- Han de definir la posibilidad de hacer asignaciones.
- Los objetos han de ser destruibles.

Requisitos adicionales (los cumplen algunos contenedores):

- Los objetos se han de poder construir sin parámetros.
- Se ha de definir el operador ==.
- Se ha de definir el operador < para permitir la ordenación.

Si se usan punteros, al utilizar el operador de igualdad no compara los elementos, sino sus direcciones de memoria. Lo mismo ocurre al ordenarlos con el operador <, los ordena según su dirección de memoria.

¿Cómo solucionamos el problema anterior?

Modificando los operadores para que comparen dos punteros a un elemento y los compare como nosotros queramos.

ejemplo: `int operador<(const Cronometro *op1, Cronometro *op2){ ... }`

Nota: cuando exista un algoritmo implementado en una función y en un método de contenedor conviene usar el método por motivos de eficiencia.

Para utilizar determinadas operaciones, los tipos deben ser iguales.

ejemplo: no se puede intercambiar un vector de enteros con uno de char.

Iteradores (en profundidad):

Son una abstracción que permite iterar por los elementos de una colección. Utilizan un patrón de diseño llamado 'iterator' que permite implementar algoritmos independientes de los contenedores.

Un objeto iterador, representa una posición dentro de un contenedor. Los iteradores tienen una interfaz reducida para mejorar y simplificar la genericidad en los algoritmos:

operador * : permite acceder al elemento al que señala el iterador.
operador ++ : avanza al siguiente elemento.
operador -> : pasa un mensaje al elemento al que señala el iterador.
operador == : true los iteradores comparados apuntan a la misma dirección de memoria.
operador != : true si son iteradores distintos.
operador = : asigna el valor de un iterador a otro.

Métodos que ofrecen los contenedores para obtener iteradores

begin: devuelve un iterador al primer elemento.
end: devuelve un iterador al elemento siguiente al último.
rbegin: devuelve un iterador al último elemento.
rend: devuelve un iterador al elemento anterior al primero.

Los métodos anteriores los poseen todos los contenedores. Además, todos los contenedores tienen dos tipos de iteradores que podemos usar: el iterador y el const_iterator.

ejemplo: `vector<int>::iterator it;`

ejemplo: implementar un método mostrar y rellenar con iteradores en dos contenedores: array y vector.

```
template <typename T>
void mostrar(T it1, T it2){
    while(it1 != it2){
        cout << *it1 << ' ';
        ++it1; //es algo más rápido que it1++
    }
}

template <typename T>
void rellenar(T it1, const T &it2, const T &v){
    while (it1!=it2){
        *it1 = v;
        ++it1;
    }
}

int main (){
    int w[] = {1,2,3,4,2,3};
    vector<int> v(10)
```

```

    for (int i=0;i<v.size();i++)
        v[i]=i;

    mostrar(w,w+sizeof(w)/sizeof(w[0]));
    mostrar(v.begin(),v.end());

    rellenar(w,w+sizeof(w)/sizeof(w[0]),0);
    rellenar(v.begin(),v.end(),-1);

    mostrar(b.rbegin(),v.rend());
}

```

Algoritmos (en profundidad)

Los algoritmos de la STL son funciones globales. Estas funciones trabajan con iteradores, para conseguir independencia del contenedor. Por motivos de eficiencia, apenas se comprueban precondiciones o se lanzan excepciones.

De esta forma, se obliga al usuario a comprobar las variables que se pasan, antes de llamar a la función.

Todos los algoritmos están fuertemente sobrecargados y muchos de ellos devuelven iteradores. Algunos de estos algoritmos son:

find: encontrar elementos.
copy: copiar elementos.
fill: rellenar un contenedor.
sort: ordenar un contenedor.
replace: reemplazar elementos.

La mayoría de los algoritmos se encuentran en el header <algorithm> y los más matemáticos se hallan en <numeric>.