Clases: Herencia y polimorfismo

Punto de información

Lenguajes basados en objetos frente a orientados a objetos

Un lenguaje ***basado* en objetos** es aquel en el cual los datos y las operaciones pueden incorporarse de tal forma que los valores de datos pueden aislarse y tenerse acceso a ellos a través de las funciones de clase especificadas. La capacidad para vincular los miembros de datos con operaciones en una sola unidad se conoce como **encapsulamiento**. En C++ el encapsulamiento es proporcionado por su capacidad de clase.

Para que un lenguaje sea clasificado como ***orientado a objetos*** debe proporcionar también herencia y polimorfismo. La ***herencia*** es la capacidad para derivar una clase de otra. Una clase derivada es un tipo de datos nuevo por completo que incorpora todos los miembros de datos y funciones miembro de la clase original con los datos y miembros de funciones nuevos únicos en sí mismos. La clase usada como la base para el tipo derivado se conoce como clase ***base*** o ***padre*** y el tipo de datos derivado se conoce como clase ***derivada*** o ***hija***.

El **polimorfismo** permite que el mismo nombre de método invoque una operación en objetos de una clase padre y una operación diferente en objetos de una clase derivada.

C++, el cual proporciona encapsulamiento, herencia y polimorfismo, es un lenguaje orientado a objetos verdadero. Debido a que C, el cual es predecesor de C++, no proporciona estas características, no es un lenguaje basado en objetos ni uno orientado a objetos.

Declaración de relaciones de herencia entre clases:

```
class Nombre_Clase_Derivada : Especificador_de_acceso Nombre_Clase_Base
{};
```

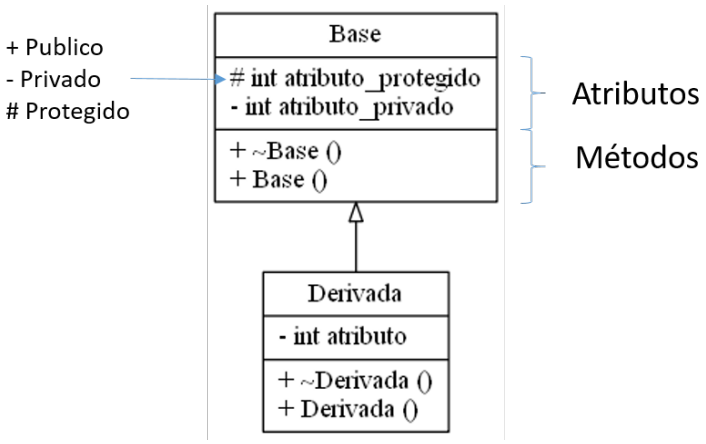
Especificador_de_acceso: public, private, protected

Tabla 10.3 Restricciones de acceso heredadas

Miembro de la clase base	Acceso de la clase derivada	Miembro de la clase derivada
private →	: private →	inaccesible
protected →	: private →	privado
public →	: private →	privado
private →	: public →	inaccesible
protected →	: public →	protegido
public →	: public →	público
private →	: protected →	inaccesible
protected →	: protected →	protegido
public →	: protected →	protegido

Ejemplo:

Diagrama de clase:



Codigo:

```
class Base
{
private:
	int atributo_privado;
protected:
	int atributo_protegido;
public:
	Base(){cout<<"Constructor Base\n";}
	~Base(){cout<<"Destructor Base\n";}

};
class Derivada: public Base
{
	int atributo;
public:
	Derivada(){cout<<"Constructor Derivada\n";}
	~Derivada(){cout<<"Destructor Derivada\n";}
};

int main() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
	Base b;
	Derivada d;
	return 0;
}
```

En este caso como la relación de herencia es publica, la clase Derivada solo puede tener accedo a los miembros públicos y protegidos de la clase Base.

Tipos de herencia:

La definición del tipo de herencia (palabra public, private o protected después de los : y antes del nombre de la clase base) determina que accesibilidad tendrán los atributos de la clase base cuando se use la clase derivada. A través de un objeto de la clase derivada es posible acceder a miembros de la clase base rompiendo la encapsulación de la clase base.

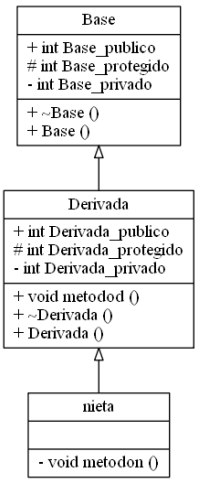
Tabla 10.3 Restricciones de acceso heredadas

Miembro de la clase base	Acceso de la clase derivada	Miembro de la clase derivada
private	: private	inaccesible
protected	: private	privado
public	: private	privado
private	: public	inaccesible
protected	: public	protegido
public	: public	público
private	: protected	inaccesible
protected	: protected	protegido
public	: protected	protegido

Es decir:

- Herencia publica: mantiene el carácter de publico privado o protegido de los miembros de la clase base.
- Herencia privada: convierte los miembros públicos y protegidos de la clase base en miembros privados de la clase derivada
- Herencia protegida: convierte en protegidos los miembros de la clase base, es decir, solo se pueden ser accedidos por métodos de clases que deriven de ella.

Para notar las diferencias probar el siguiente ejemplo variando el tipo de herencia en la clase Derivada y observar que errores devuelve el compilador:




```
class Base{
private:
	int Base_privado;
protected:
	int Base_protegido;
public:
	int Base_publico;
```

```
	Base(){cout<<"Constructor Base\n";}
	~Base(){cout<<"Destructor Base\n";}};

class Derivada:  protected Base{
private:
	int Derivada_privado;
protected:
	int Derivada_protegido;
public:
	int Derivada_publico;
	Derivada(){cout<<"Constructor Derivada\n";}
	~Derivada(){cout<<"Destructor Derivada\n";}
	void metodod(){
		Base_privado++;
		Base_protegido++;
		Base_publico++;
	}};


class nieta: public Derivada    {
	void metodon(){
		Base_privado++;
		Base_protegido++;
		Base_publico++;
		Derivada_privado++;
		Derivada_protegido++;
		Derivada_publico++;}};

int main() {
	Base b;
	b.Base_publico++;
	Derivada d;
	cout<<d.Base_protegido;
	nieta n;
	cout<<n.Base_publico;
	return 0;
}
```

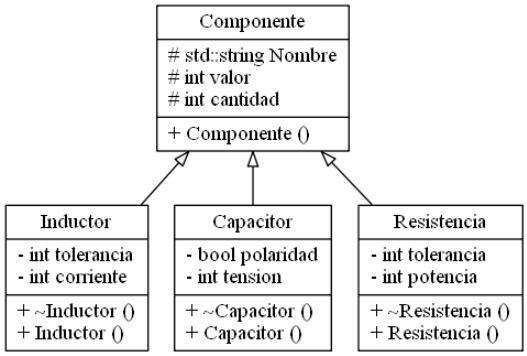
Ejercicios:

Clase Círculo:

1) Implemente en C++ la clase base **Circulo** con un atributo ***radio***, el método ***getArea()*** y otros métodos necesarios.
2) Implemente la clase **Cilindro**, derivada de **Circulo**, que incluye un atributo ***altura*** y el método ***getVolumen()***
3) Implemente la clase **Esfera**, derivada de **Circulo**, que incluya su método ***getVolumen()***
4) Implemente un programa principal en el cual se cree 1 objeto de cada clase y muestre el valor del área de cada uno y el volumen del cilindro y esfera.
5) Implemente un programa principal que permita cargar un vector de 10 objetos circulares que podrán ser de tipo **Cilindro** o **Esfera.** Luego recorra el vector mostrando el área y volumen de cada uno.
6) Convertir la clase Circulo en abstracta

Clase Componente:

1) Implemente la siguiente jerarquía de clases para gestionar una lista de componentes electrónicos:



2) Cree un programa que permita cargar un listado de componentes y luego muestre un listado de todos los componentes cargados ordenados por tipo

Clase Círculo:

Interfaces:

```
class Circulo {
protected:
	float radio;
public:
	float getRadio(){return radio;}
	Circulo(float r=0){ cout<<"Constructor circulo. \n";
		radio = r; }
	virtual ~Circulo(){cout<<"Destructor Circulo.\n";}
	float getArea(void){return 3.14f * radio * radio;}
	float getVolumen(void){return 0;};
};


class Cilindro: public Circulo{
	float altura;
public:
	Cilindro(float h = 0, float r = 0) : Circulo(r)
	{	cout<< "Constructor cilindro\n";
		altura = h;
	}
	float getVolumen(void)
	{	return getArea()*altura;   }
	~Cilindro(){cout<< "Destructor cilindro\n";}
};


class Esfera: public Circulo {
public:
	Esfera(float r=0):Circulo(r){cout<<"Constructor esfera\n";}
	~Esfera(){cout<<"Destructor esfera\n";}
	float getVolumen(){return 3.14f *4/3*radio*radio*radio;}
};
```

Comentado [VL1]: Aquí llamo al constructor de la clase Circulo para que se cree el objeto circulo con el radio adecuado. Esto es encadenar constructores

Programa principal:

```
int main() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!

	Circulo c1(2);	//Creo un objeto de la clase Circulo
	cout<<c1.getRadio()<<c1.getVolumen()<<endl;

	Cilindro cl1(3,2); //Creo un objeto de la clase Cilindro
	cout<<"Volumen cilindro: "<<cl1.getVolumen()<<"Area:"<<cl1.getArea()<<endl;

	Esfera e(4); //Creo un objeto de la clase Esfera
	cout<<"Volumen esfera: "<<e.getVolumen()<<endl; //Volumen de la esfera
	cout<<"Volumen del circulo de esfera:"<<e.Circulo::getVolumen()<<endl; //Desde un objeto
esfera puedo acceder al método getVolumen del circulo que forma parte del objeto esfera
	return 0;
}
```

Salida del programa:

```
!!!Hello World!!!
Constructor circulo.
20
Constructor circulo.
Constructor cilindro
Volumen cilindro: 37.68 Area: 12.56
Constructor circulo.
Constructor esfera
Volumen esfera: 267.947
Volumen del circulo de esfera: 0
Destructor esfera
Destructor Circulo.
Destructor cilindro
Destructor Circulo.
Destructor Circulo.
```

Comentado [VL2]: Observar que se ejecuta primero el constructor de la clase base

Comentado [VL3]: Desde un objeto de una clase derivada puedo acceder a métodos de la clase base, inclusive si tienen el mismo nombre (métodos sobrecargados)

Comentado [VL4]: Los destructores se ejecutan en orden inverso

Vector de objetos: Si deseo cargar un vector que pueda contener objetos de distintas clases dentro de la estructura de herencia (Circulo, Cilindro y Esfera) puedo utilizar una vector de punteros a la clase base (Circulo) e ir asignando a cada puntero del vector cada objeto creándolos de forma dinámica (new, malloc).

```
Circulo *p[10];


int opcion = 0;
for(int i = 0; i<3; i++)
{
	cout<<"Que desea cargar: ";
	cin>>opcion;
	switch(opcion){
	case 1: //Circulo
		p[i] = new Circulo(2); break;
	case 2: //Cilindro
		p[i] = new Cilindro(3,2);break;
	case 3: //Esfera
		p[i] = new Esfera(2);break;
	}
}
```

Recorrido del vector y mostrar el volumen de cada objeto:

```
for(int i = 0; i<3; i++)
	{
		cout<<"Volumen: "<<p[i]->getVolumen()<<endl;
	}
```

Salida:

```
!!!Hello World!!!
Que desea cargar: 1
Constructor circulo.
Que desea cargar: 2
Constructor circulo.
Constructor cilindro
Que desea cargar: 3
Constructor circulo.
Constructor esfera
Volumen: 0
Volumen: 0
Volumen: 0
```

Comentado [VL5]: Observar que 0 es el volumen de Circulo, por ende no se esta calculando de manera correcta el volumen para objetos Cilindro y esfera

Polimorfismo: Para que el programa sea capaz de escoger la funcion getVolumen adecuada debemos usar polimorfismo. El polimorfismo en tiempo de ejecución es logrado por una combinación de dos características: 'Herencia y funciones virtuales".

Una función virtual es una funcion que es declarada como 'virtual' en una clase base y es redefinida en una o más clases derivadas. Además, cada clase derivada puede tener su propia versión de la funcion virtual. Lo que hace interesantes a las funciones virtuales es que sucede cuando una es llamada a través de un puntero de clase base (o referencia). En esta situación, C++ determina a cual versión de la funcion llamar basándose en el tipo de objeto apuntado por el puntero. Y, esta determinación es hecha en 'tiempo de ejecución'. Además, cuando diferentes objetos son apuntados, diferentes versiones de la funcion virtual son ejecutadas. En otras palabras es el tipo de objeto al que está siendo apuntado (no el tipo del puntero) lo que determina cual versión de la funcion virtual será ejecutada. Además, si la clase base contiene una funcion virtual, y si dos o más diferentes clases son derivadas de esa clase base, entonces cuando tipos diferentes de objetos están siendo apuntados a través de un puntero de clase base, diferentes versiones de la funcion virtual son ejecutadas. Lo mismo ocurre cuando se usa una referencia a la clase base.

Se declara una funcion como virtual dentro de la clase base precediendo su declaración con la palabra clave virtual. Cuando una funcion virtual es redefinida por una clase derivada, la palabra clave 'virtual' no necesita ser repetida (aunque no es un error hacerlo).

Una clase que incluya una funcion virtual es llamada una 'clase polimórfica'. Este término también aplica a una clase que hereda una clase base conteniendo una funcion virtual.

Nueva definición de la clase Círculo:

```
class Circulo {
protected:
      float radio;
public:
      float getRadio(){return radio;}
      Circulo(float r=0){ cout<<"Constructor circulo. \n";
            radio = r; }
      virtual ~Circulo(){cout<<"Destructor Circulo.\n";}
      float getArea(void){return 3.14f * radio * radio;}
      virtual float getVolumen(void){return 0;};
};
```

Ahora la salida es:

```
!!!Hello World!!!
Que desea cargar: 1
Constructor circulo.
Que desea cargar: 2
Constructor circulo.
Constructor cilindro
Que desea cargar: 3
Constructor circulo.
Constructor esfera
Volumen: 0
Volumen: 37.68
Volumen: 33.4933
```

Comentado [VL6]: Observar que ahora se muestra correctamente el volumen de cada tipo de objeto

Funcion virtual pura:

Dado que no tiene sentido hablar de volumen de un circulo, puedo declarar a la funcion getVolumen de la clase circulo como virtual pura. Una funcion virtual pura es una funcion declarada en la

clase base pero que no tiene definición (código) en la misma y, por lo tanto, no se puede ejecutar. Para declarar una funcion virtual pura debo asignarle 0 en la clase base:

```
class Circulo {
protected:
      float radio;
public:
      float getRadio(){return radio;}
      Circulo(float r=0){ cout<<"Constructor circulo. \n";
            radio = r; }
      virtual ~Circulo(){cout<<"Destructor Circulo.\n";}
      float getArea(void){return 3.14f * radio * radio;}
      virtual float getVolumen(void)=0;
};
```

Comentado [VL7]: Funcion virtual pura

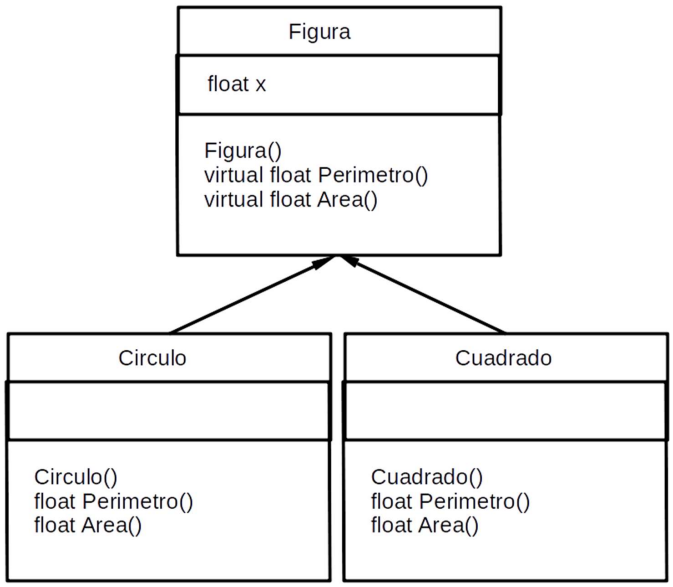
Dado que la clase Circulo contiene un método que no está definido, no es posible utilizar objetos de esa clase. Se dice que la clase Circulo es una clase abstracta.

Si una clase tiene al menos una funcion virtual pura, entonces esa clase se dice que es 'abstracta'. Una clase abstracta tiene una característica importante: No puede haber objetos de esa clase. En vez de eso, una clase abstracta debe ser usada solo como una base que otras clases heredaran. La razón por la cual una clase abstracta no puede ser usada para declarar un objeto es, por supuesto, que una o más de sus funciones no tienen definición. Sin embargo, incluso si la clase base es abstracta, la puede usar aun para declarar punteros o referencias, los cuales son necesarios para soportar el polimorfismo en tiempo de ejecución.

Referencia:
https://es.wikibooks.org/wiki/Programación_en_C%2B%2B/Funciones_virtuales#FUNCIONES_VIRTUALES_PURAS_Y_CLASES_ABSTRACTAS

Otros ejercicios:

1) Implemente la siguiente estructura de clases, tanto su interface como los métodos indicados.



Implemente un programa principal que permita la carga de un arreglo de figuras, cuyo tamaño será definido por el usuario en tiempo de ejecución (memoria dinámica). Luego de la carga, el programa deberá mostrar en pantalla el perímetro y área de cada figura cargada.

2) Crear una clase Matriz que permita almacenar elementos de matrices de tamaño NxM de tipo double. Implemente los métodos constructor, destructor y los necesarios para cargar los valores de la matriz y mostrarlos en pantalla (sobrecarga de operadores <<, >> o métodos). Finalmente sobrecargue los operadores + y * los cuales deben incluir una verificación de compatibilidad de los tamaños de cada matriz (igual tamaño para suma y nro de columnas de la primera matriz igual a número de filas de la segunda para el producto), en caso de no cumplirse deben devolver una matriz nula (tamaño cero). Determinar los atributos que debe incluir la clase.

3) Descargue y compile la librería CppLinuxSerial:
(https://github.com/gbmhunter/CppLinuxSerial)
   a. Estudie la definición de la interface de la clase:
      https://github.com/gbmhunter/CppLinuxSerial/blob/master/include/CppLinuxSerial/SerialPort.hpp
   b. Estudie la implementación de los métodos:
      https://github.com/gbmhunter/CppLinuxSerial/blob/master/src/SerialPort.cpp
   c. Pruebe el ejemplo que envía un Hello y lo recibe.
   d. Modifíquelo para recibir un Hello desde un Arduino u otro programa en C/C++ si no dispone de una placa

Herencia múltiple, herencia en diamante y clases virtuales

Herencia multiple: cuando una clase deriva de 2 o mas clases base.

Ejemplo:

```
class ClaseA{public: ClaseA(){cout<<"Constructor ClaseA\n";}};

class ClaseB: public ClaseA{public: ClaseB(){cout<<"Constructor ClaseB\n";}};
class ClaseC: public ClaseA{public: ClaseC(){cout<<"Constructor ClaseC\n";}};

class ClaseD: public ClaseB, public ClaseC{public: ClaseD(){cout<<"Constructor
ClaseD\n";}};

int main() {
ClaseD d;
return 0;
}
```

Comentado [VL8]: Herencia simple

Comentado [VL9]: Herencia multiple, listar las clases bases luego del : separando cada una por comas

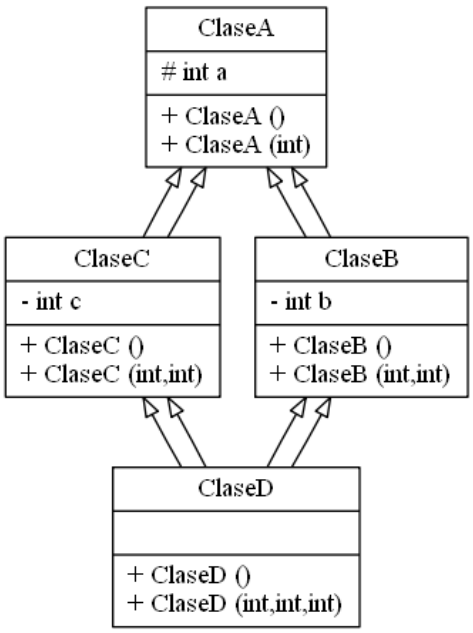
Comentado [VL10]: Al crear un objeto de la clase Nieta se van a crear previamente los objetos de cada clases base

Constructores en herencia multiple:

Recorder que al crear un objeto de una clase que deriva de otra, primero se va a crear un objeto de la clase base, por ende el contructor de la clase base se ejecuta antes del de la derivada. Es posible pasar parámetros al constructor de la clase base encadenando constructores al implementar los constructores de la clase derivada mediante el operador ":".

```
class ClaseA{protected: int a; public: ClaseA(int i){a = i; cout<<"Constructor
ClaseA\n";}};

class ClaseB: public ClaseA{int b;
public: ClaseB(int i, int j):ClaseA(i){b = j; cout<<"Constructor ClaseB\n";}};


class ClaseC: public ClaseA{int c; public: ClaseC(int i, int k):ClaseA(i){c = k;
cout<<"Constructor ClaseC\n";}};

class ClaseD: public ClaseB, public ClaseC{public:
      ClaseD(int i, int j, int k):ClaseB(i,j),ClaseC(i,k){cout<<"Constructor
ClaseD\n";}};


int main() {
      ClaseD d(1,2,3);
      return 0;
}
```

El ejemplo anterior representa el problema de herencia en diamante:



En este caso, al crear un objeto de la ClaseD se van a terminar creando dos objetos independientes de la ClaseA, cada uno con sus propios atributos. Si se desea evitar esta situación y que solo se genere un único objeto de la ClaseA es necesario declarar la herencia de ClaseC y ClaseB como "`virtual`". En este caso, es necesario llamar explícitamente al constructor de la ClaseA en la ClaseD o se ejecutará el constructor por defecto de la misma si lo tuviere.

```
class ClaseA{protected: int a; public: ClaseA(int i){a = i; cout<<"Constructor
ClaseA\n";}};

class ClaseB: virtual public ClaseA{int b;
public: ClaseB(int i, int j):ClaseA(i){b = j; cout<<"Constructor ClaseB\n";}};


class ClaseC: virtual public ClaseA{int c; public: ClaseC(int i, int k):ClaseA(i){c =
k; cout<<"Constructor ClaseC\n";}};

class ClaseD: public ClaseB, public ClaseC{public:
      ClaseD(int i, int j, int k):ClaseB(i,j),ClaseC(i,k),ClaseA(i){cout<<"Constructor
ClaseD\n";}};
```

Archivos Makefile

Son scripts (archivo de texto con comandos de consola) que permiten automatizar y personalizar el proceso de compilación y desarrollo de un software. Como minimo incluyen la línea del comando de compilador necesaria para compilar y generar un archivo ejecutable a partir del/os archivo/s de código del programa.

Es un archivo de texto con el nombre “Makefile” que se puede editar en cualquier programa de edición de textos.

Contiene una o más reglas a ejecutar, cada una indicada con un nombre seguida de dos puntos (:).

Para compilar un programa mediante archivos makefiles se utiliza el comando `make` seguido del nombre de la regla que se desea ejecutar:

Para ejecutar la primer regla:

- `make`

Ejecutar regla contador:

- `make contador`

Ejecutar regla clean:

- `make clean`

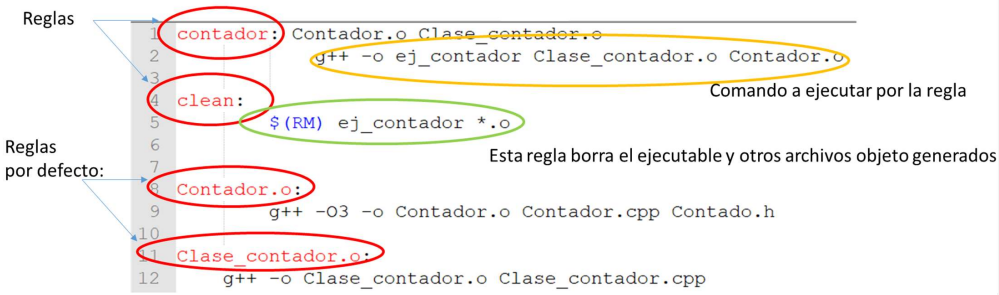
Ejemplo básico:

Se tiene el programa Contador integrado por los archivos Contador.cpp, Contador.h y Clase_contador.cpp.

Archivo Makefile minimo:

```
contador:
    g++ -o ej_contador Clase_contador.cpp Contador.cpp
```

Makefile con mas reglas:



Otro makefile que permite mediante etiquetas, elegir opciones de compilación:

```
# the compiler: gcc for C program, define as g++ for C++
CC = g++

# compiler flags:
#  -g    adds debugging information to the executable file
#  -Wall turns on most, but not all, compiler warnings
CFLAGS  = -g -Wall

# the build target executable:
TARGET = ej_contador

all: $(TARGET)

$(TARGET): Clase_contador.o Contador.o
	$(CC) $(CFLAGS) -o $(TARGET) Clase_contador.o Contador.o

clean:
	$(RM) $(TARGET) *.o *~

contador.o: Contador.cpp Contador.h
	$(CC) $(CFLAGS) -c Contador.cpp

Clase_contador.o: Clase_contador.cpp Contador.h
	$(CC) $(CFLAGS) -c Clase_contador.cpp
```