Práctico de Programación 4 – clase 12

Práctico 6, Ejercicio 2:

El código presentado en este ejercicio no es habitual en programas C++. Sin embargo, aquí es utilizado para analizar en profundidad los mecanismos de herencia, polimorfismo, sobrecarga de funciones, casteo, sustitución, sobreescritura (u overriding) y ligadura dinámica (o binding dinámico), que implementa el lenguaje.

Código	*Invocaciones*	*Observaciones*
`claseA *a, *ap;`		Se reserva memoria estática en el stack[1] para dos punteros a claseA
`claseB *b, *bp;`		Idem. anterior para punteros a claseB
`claseC *c;`		Idem. anterior para un puntero a claseC
`claseD *d;`		Idem. anterior para un puntero a claseD
`int in = 0;`	Constructor de copia para el tipo elemental int	
`short sh = 4;`	Idem. anterior para el tipo elemental short (entero corto)	
`a = new claseA();`	Operador new para un objeto de claseA Asignación de punteros	Se reserva memoria dinámica en el heap[2] para una variable de claseA
`b = new claseB();`	Idem. anterior para claseB	Idem. anterior para claseB
`c = new claseC();`	Idem. anterior para claseC	Idem. anterior para claseC
`d = new claseD();`	Idem. anterior para claseD	Idem. anterior para claseD
`claseB bb;`	Constructor por defecto de claseB	Se reserva memoria estática para una variable de claseB
`bb = (*c);`	Operador de asignación de claseB, equivalente a bb.operator=(*c)	Se toma solamente la parte correspondiente a el objeto apuntado por c (casteo de claseC a claseB)
`bb.f1(in);`	f1 de claseB	Ligadura estática (se decide en tiempo de compilación) dado que se invoca a través de un objeto (no puntero * ni referencia &)
`ap = b;`	Sustitución (un puntero a un objeto de una clase puede apuntar a cualquier objeto de una clase derivada de ella)	ap (puntero a claseA) pasa a apuntar a un objeto de claseB
`bp = c;`	Análogo a anterior	bp (puntero a claseB) pasa a apuntar a un objeto de claseC

[1] Espacio de memoria estática, local al bloque donde se declara la variable. Se libera automáticamente al finalizar el bloque.

[2] Espacio de memoria dinámica. Se obtiene con new y se libera con delete.

`ap->f1(in);`	f1 de claseA	Por más que ap apunte a un objeto de claseB, f1 no es virtual en claseA, por lo tanto su ligadura es estática (el método a invocar se decide en tiempo de compilación)
`ap->f3(in);`	f3 de claseB	f3 es virtual en la clase del tipo estático (del cual se declaró ap, es decir claseA), entonces su invocación se decide en tiempo de ejecución (ligadura dinámica) en base al tipo dinámico (el tipo del objeto al que apunta actualmente, es decir claseB). Notar que claseB sobreescribe a f3 (idéntico cabezal que en claseA)
`bp->f3(in);`	f3 de claseC	Análogo al caso anterior. El tipo estático es claseB, f3 es virtual (aunque no explícitamente en claseB, queda definido desde claseA) y el tipo dinámico es claseC, donde la función está sobreescrita
`bp->f1(in);`	f1 de claseC	También análogo. El tipo estático es claseB, f1 es virtual (explícitamente en claseB) y el tipo dinámico es claseC, donde la función está sobreescrita
`bp->f2(in);`	f2 de claseB	El tipo estático es claseB, f2 es virtual en claseB, el tipo dinámico es claseC, pero ahí f2 NO está sobreescrita (recibe un short en lugar de un int)
`ap = c;`	Sustitución (un puntero a un objeto de una clase puede apuntar a cualquier objeto de una clase derivada de ella)	ap (puntero a claseA) pasa a apuntar a un objeto de claseC
`ap->f2(in);`	f2 de claseC	El tipo estático es claseA, se castea la variable de tipo int a short (para encontrar una función que se pueda invocar con ese parámetro), f2 es virtual en claseA, el tipo dinámico es claseC y sobreescribe f2
`ap->f2(sh);`	f2 de claseC	Similar al caso anterior, pero sin casteo dado que la variable ya es de tipo short
`ap = b;`	Sustitución (un puntero a un objeto de una clase puede apuntar a cualquier objeto de una clase derivada de ella)	ap (puntero a claseA) pasa a apuntar a un objeto de claseB
`ap->f2(sh);`	f2 de claseA	El tipo estático es claseA, f2 es virtual en claseA, el tipo dinámico es claseB, pero NO sobreescribe f2 (el parámetro es de tipo int en lugar de short)
`ap = d;`	Sustitución (un puntero a un objeto de una clase puede apuntar a	ap (puntero a claseA) pasa a apuntar a un objeto de claseD

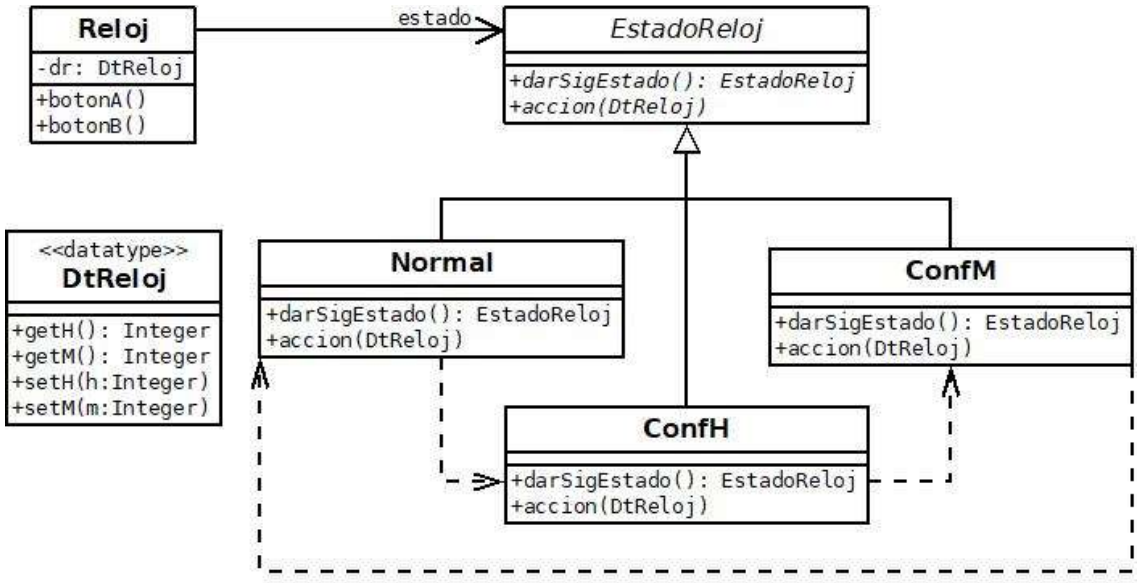
	cualquier objeto de una clase derivada de ella)	
`ap->f1(in);`	f1 de claseA	El tipo estático es claseA y f1 no es virtual, entonces la ligadura es estática (la invocación se decide en tiempo de compilación) y no importa el tipo dinámico
`a->f1(in);`	f1 de claseA	Misma explicación que la línea anterior

Práctico 6, Ejercicio 12:

Partimos del diseño dado por el siguiente diagrama, donde se aplicó el patrón State (ver clase de teórico *15 - Diseño: Patrones de Diseño*) con los siguientes roles:

- Reloj es el Contexto. La operación botonB es la solicitud.
- EstadoReloj es el Estado. La operacion accion es el evento.
- Las clases Normal, ConfH y ConfM son los estados concretos.

Adicionalmente, la operación botonA es la que realiza los cambios de estados.




```
class Reloj {
      private:
            DtReloj dr;
            EstadoReloj *er;
      public:
            Reloj();
            ~Reloj();
            void botonA();
            void botonB();
};
```

```
Reloj::Reloj() {
      er = new Normal;
      // Asumimos que dr se inicializa por defecto con
      // horas y minutos en cero
}

Reloj::~Reloj() {
      delete er;
}

void Reloj::botonA() {
      EstadoReloj *aux = er->darSigEstado(); // Pido el siguiente estado
                                             // al estado actual
      delete er; // Elimino el estado actual
      er = aux; // Asigno el siguiente estado
}

void Reloj::botonB() {
      er->accion(dr); // Invoca la accion del estado actual
                      // Pasando como parámetro (por referencia)
                      // a los datos del reloj para que puedan modificarse
}

class EstadoReloj {
      public:
            virtual EstadoReloj *darSigEstado() = 0;
            virtual void accion(DtReloj &) = 0;
};

class Normal : public EstadoReloj{
      public:
            virtual EstadoReloj *darSigEstado();
            virtual void accion(DtReloj &);
};

EstadoReloj *Normal::darSigEstado() {
      return new ConfH; // Devuelve el siguiente estado según el diagrama
}

void Normal::accion(DtReloj &dr) {
      // No hace nada en este estado (por definición)
}

class ConfH : public EstadoReloj{
      public:
            EstadoReloj *darSigEstado();
            void accion(DtReloj &);
};

EstadoReloj *Normal::darSigEstado() {
      return new ConfM; // Devuelve el siguiente estado según el diagrama
}

void Normal::accion(DtReloj &dr) {
      dr.setH((dr.getH()+1) % 24);
      // Incremento la hora en una unidad
      // Si termina el día vuelvo a 0 mediante el resto de la división entera
```

```
        // El parámetro dr sale modificado (se pasa por referencia)
}
```

La clase **ConfM** es análoga a ConfH con las siguientes diferencias:

- El siguiente estado es Normal.
- El incremento de minutos debe implementar la lógica correcta si se llega a 60 unidades. A su vez, se debe verificar si se llegó al final del día.

Observaciones:

- En esta implementación se asume que al cambiar de estado se destruye el estado actual (lo hace el Reloj/Contexto en la operación botonA) y se crea el nuevo (lo hace el EstadoReloj/Estado en la operación darSigEstado).
- Una alternativa sería implementar los estados mediante el patrón Singleton. En ese caso, al cambiar de estado el Contexto NO debe destruir el Estado actual y la operación darSigEstado no debe instanciar un nuevo estado sino pedir la única instancia que existe al estado que corresponda.