FIEC

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN

M.Sc. Alexander Prieto León

Conferencia VI

Unidad II Programación orientada a objetos y estructuras de datos aplicadas a automatización

2.2 Clases en C++ y su composición.

Objetivos:

- Conceptualizar el paradigma de la Programación Orientada a Objetos.
- Identificar los roles en un programa.
- Implementar el encapsulamiento en C++.
- Implementar clases y sus funciones miembros.
- Implementar los constructores y el destructor de una clase.
- Implementar clases compuestas.

BIBLIOGRAFÍA

Bibliografía:

- Deitel and Deitel. Como programar en C/C++. Segunda edición o superior.
- García de Jalón, J.; y otros. Aprenda C++ como si estuviera en primero. Universidad de Navarra.

- **En la clase anterior:**

- **Preguntas iniciales:**
  - ¿Qué almacena un **struct**?

- Los objetos son **Tipos Abstractos de Datos**.
- **Buscan modelar el espacio del problema a través del propio lenguaje de programación.**

- Contienen características y funcionalidades.

Tipo de objeto	**FOCO**
Características	bool encendido int intensidad
Funcionalidades (Interfaz)	encender() apagar() brillar() atenuar()

Inv Luminarias
encendido → 1 → 0
intensidad → ↑ → ↓

Private

Public

Características de la POO:

- **Todo es un objeto.** Piense en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.
- **Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes.** Para hacer una petición a un objeto, basta con "enviarle un mensaje". Más concretamente, puede considerarse que un mensaje en sí es una petición para solicitar una llamada a una función que pertenece a un objeto en particular.

Características de la POO:

- **Cada objeto tiene su propia memoria, constituida por otros objetos.** Dicho de otra manera, uno crea una nueva clase de objeto construyendo un paquete que contiene objetos ya existentes. Por consiguiente, uno puede incrementar la complejidad de un programa, ocultándola tras la simplicidad de los propios objetos.

- **Todo objeto es de algún tipo.** Cada objeto es un elemento de una clase, entendiendo por "**clase**" un sinónimo de "**tipo**". La funcionalidad de una clase la constituyen "el conjunto de mensajes que se le pueden enviar".

Características de la POO:

- **Todos los objetos de determinado tipo pueden recibir los mismos mensajes.** Ésta es una afirmación de enorme trascendencia como se verá más tarde. Dado que un objeto de tipo "cuadrado" es también un objeto de tipo "rectángulo", se garantiza que todos los objetos "cuadrado" acepten mensajes propios de "rectángulo". Esto permite la escritura de código que haga referencia a rectángulos, y que de manera automática pueda manejar cualquier elemento que encaje con la descripción de "rectángulo". Esta capacidad de suplantación es uno de los conceptos más potentes de la POO.

Roles en un programa:

- Estructura para un tipo de datos que modela la hora del día:

```
struct STime{
    int hour;
    int minute;
    int second;
};
```

La utilización de esta clase o estructura implica tres roles importantes que serán definidos:

1-Programador de la Clase(PC): Es quien define la **clase** con sus propiedades. (En este caso define la **struct** Stime)

2-Programador Usuario de Clases(PUC): Es el programador que usa la **clase** o Tipo de dato **struct** para construir programas (define objetos o variables en el programa y los usa).

En algunas ocasiones 1 y 2 convergen en la misma persona.

- **Roles en un programa:**

```
struct STime{
    int hour;
    int minute;
    int second;
};
```

2-Programador Usuario de Clases(PUC):

Ejemplo:

```
STime h1;
h1.hour = 10; h1.minute = 25; h1.second = 44;
```

3-Usuario de programa (UP): Es quien usa el programa compilado.

Necesidad del encapsulamiento.

- ¿Qué problemas presenta esta definición?

Un *Programador Usuario de Clase* (PUC) puede crear objetos de tipo STime inconsistentes, ya que nada le impide hacer la siguiente asignación que viola el concepto de Hora:

```
STime h1;
h1.hour = 99; h1.minute = 88; h1.second = 77;
```

- ¿Cómo se resuelve este problema?

Encapsulación: Permite lograr la consistencia adecuada del modelo que se desea programar y es de especial utilidad para los PUC.

- Definición de una **clase** (Rol PC):

```
class CTime{
public:
     CTime(int, int, int);
     void setTime(int, int, int);
     void printMilitary();
     void setHour(int h)
    { hour = (h >= 0 && h < 24)? h:0;}
     int getHour() const{return hour;}

private:
    int hour;
    int minute;
    int second;
};
```

```
CTime::CTime(int h=0, int m=0, int s=0){
    hour   = (h >= 0 && h < 24)? h:0;
    minute = (m >= 0 && m < 60)? m:0;
    second = (s >= 0 && s < 60)? s:0;
}

void CTime::setTime(int h, int m, int s) {
hour   = (h >= 0 && h < 24)? h:0;
minute = (m >= 0 && m < 60)? m:0;
second = (s >= 0 && s < 60)? s:0;
}

void CTime::printMilitary() {
cout << (hour   < 10? "0": "" ) << hour
       << ":"<< (minute < 10? "0": "" )
       << minute << ":"
       << (second<10?"0": "" ) << second;
}
```

- Observaciones:
  - Nuevos conceptos:
    - Miembros de una clase: Propiedades o atributos (ídem a estructuras)
    - Miembros de una clase: Métodos: Son las operaciones que se pueden aplicar sobre las clases que se define.
  - Nuevas palabras claves:
    - **public:** Define un miembro público al cual los PUC tienen libre acceso.
    - **private**: Define un miembro privado al cual los PUC no tienen acceso. Solo los PC tienen acceso a estos campos.
  - Nuevos sentidos a los operadores:
    - "::" Permite definir a que clase pertenece un método de clase (función que es miembro de una clase).
  - Los métodos u operaciones de clases pueden acceder libremente a los miembros privados y públicos de una clase.

- Observaciones:
  - **setHour** y **getHour** son métodos **inline:** Se utilizan cuando la definición es muy elemental. Funcionan como macros.
  - **const** Utilizado al final del prototipo de un método indica que el método es aplicable a objetos creados con el modificador **const**
  - Existen dos métodos especiales que deben ser analizados por parte del PC.
    - **Constructor.**
    - **Destructor.**

- Observaciones:
  - **Constructor**: Se define para construir correctamente un objeto de la clase.
    - Tienen el mismo identificador que el usado para la clase
    - Se invoca en el momento en que se crea un objeto de la clase.
    - No se les especifica valor de retorno al declararlos.
    - Valor implícito para un parámetro, si éste es omitido al invocar a la función.
    - Se pueden declarar "homónimos" (varios constructores)
    - Existen varios tipos de constructores:
      - de copia: Permite asignar objetos de clase entre sí en el momento de la creación. C++ proporciona uno por defecto que copia bit a bit la memoria de un objeto hacia el otro.
      - Por defecto u por omisión: permite crear objetos con valores específicos por defecto.

- Observaciones:

  - **Destructor**:
    - tiene el mismo nombre que la clase en la que se define, pero poniéndole como prefijo el símbolo tilde (**~**), en este caso podría ser: ~CTime() en la definición de la clase,
    - se declara siempre con la lista de parámetros **vacía.**
    - No es posible definirle homónimos.
    - No se le especifica valor de retorno al declararlo
    - Se invoca automáticamente antes de que un objeto sea destruido.
    - Si antes de que se destruya un objeto de una clase no hay que realizar operación alguna, entonces para esa clase no se definirá el método destructor.

- Programa principal(Rol PUC):

```
int main() {
CTime t1(3,36,45);      // creación de un objeto y llamada al constructor.
t1.printMilitary();     // llamada de un miembro público de clase (método)
t1.setTime(8,2,23);     // llamada de un miembro público de clase (método)

CTime t2;               // llamada al constructor con parámetros por defecto
t2.printMilitary();

const CTime t3 = t2;    // llamada al constructor de copia: const CTime t3(t2)
t3.printMilitary();     //llamada a un método no aplicable a objetos const, genera
                        // WARNING o ERROR según el compilador que se utilice
t3.getHour();           // llamada a un método aplicable a objetos const

t3. setHour(3);         // llamada a un método no aplicable a objetos const, genera un
                        // ERROR por parte del compilador alertando al PUC que
                        // está realizando una operación incorrecta
return 0;
}
```

Composición de clases:

- Es un concepto similar al que ya se conoce de las estructuras: Una clase (**externa**) está compuesta por una o varias clases (**internas**) si estas contiene miembros que a su vez han sido definidos previamente como clases.
- Siguiendo la misma idea de *CTime* podríamos escribir una clase similar para la Fecha:

```
class Date {
public:
  Date(int=1,int=1,int=1900); // constructor por defecto consistente
  void print() const;
private:
  int month;
  int day;
  int year;
};
```

Composición de clases:

- Luego una clase empleado puede estar compuesta por su nombre, apellido, fecha de nacimiento y fecha de contrato. Esto implicará que 2 de los miembros a su vez sean también clases.

```
class Employee {
public:

Employee(char*,char*,int,int,int,int,int,int);
 Employee(const Employee&);
 ~Employee();
private:
  char* lastname;
  char* firstname;
  Date birthDate;
  Date hireDate;
};
```

```
Employee::Employee(char* fname,char* lname,
                   int bm, int bd, int by,
                   int hm, int hd, int hy)
        :birthDate(bm,bd,by),
         hireDate(hm,hd,hy)
{
  firstname=new char[strlen(fname)+1];
  strcpy(firstname, fname);
  lastname=new char[strlen(lname)+1];
  strcpy(lastname, lname);
}
```

Inicialización en Constructor:

- El constructor de la clase empleado es el responsable de construir correctamente los objetos Date que son sus miembros. Para ello se utilizan los inicializadores que no es más que una llamada a los constructores de las clases internas utilizando el operador de inicialización **":"**. Los parámetros pasados a los constructores internos se toman de los parámetros del constructor externo. Esta es la única forma de inicializar miembros constantes de una clase (definidos en la clase con el calificador **const** delante).

- **Nota:** se construyen los objetos miembro, antes de que los objetos de clase que los incluyen sean construidos. Si un objeto tiene varios objetos miembro, está indefinido el orden en el cual los objetos miembro serán construidos.

Composición de clases:

- Una variable de tipo empleado se declara de la siguiente forma:

```
Employee e("Pedro","Perez", 5,24,1980,9,3,2004);
```

- o dinámicamente:

```
Employee* Pe = new Employee ("Pedro","Perez", 5,24,1980,9,3,2004);
:
:
delete Pe;
```

Composición de clases:

- Observaciones:
  - Cualquier método de clase puede acceder libremente a las propiedades y métodos públicos de *birthDate* y *hireDate*. Este es el caso del Rol de PC.
  - Los PUC nunca accederán directamente a *birthDate*, *hireDate*, así como tampoco a *firstname* y *lastname*. Para ello el PC debe crear los métodos apropiados.

Constructor de copia:

- Como se crean los datos del nombre y el apellido de forma dinámica el constructor de copia que crea el compilador automáticamente ya no funciona adecuadamente puesto que una copia **byte a byte** del objeto no copiaría el nombre y el apellido sino que copiaría el puntero al nombre y el apellido del objeto original. Esto no sólo es un error conceptual sino que puede acarrear errores de ejecución si se destruye correctamente uno de los dos objetos y el otro intenta acceder al nombre que ya no existe. Por ello el constructor de copia debe hacer una nueva reserva de memoria para el nombre y el apellido, y copiar allí los del original:

Constructor de copia:

```
Employee::Employee(const Employee& E)
{
   birthDate = E.birthDate;
   hireDate = E.hireDate;
   firstname=new char[strlen(E.firstname)+1];
   strcpy(firstname, E.firstname);
   lastname=new char[strlen(E.lastname)+1];
   strcpy(lastname, E.lastname);
}
```

Observaciones:

- El parámetro se pasa como referencia constante que es una versión más eficiente que el paso por valor. En este caso no se puede hacer un paso por valor convencional porque éste, para crear la copia del valor, llama precisamente al constructor de copia, y se convertiría en un llamado recursivo infinito.

- Fíjese que se pudiera pensar que se está realizando una violación del control de acceso a miembro privado al llamar por ejemplo a **E.birthDate** o cualquiera de los otros miembros privados de E, pero la realidad es que se sigue en el **Rol PC**, por tanto, se tiene acceso a todos los elementos privados de la clase.

- En este caso no se llama con "**:**" a los inicializadores de **birthDate** y **hireDate** puesto que no hace falta comprobar si los valores son correctos pues ya fueron comprobados al ser introducidos en el objeto original.

Constructor de conversión:

- Entre otros tipos de constructores vale la pena mencionar el **constructor de conversión** que es el que se llama cuando se necesita convertir de un tipo de dato recibido como argumento a otro tipo de dato que es el del objeto a construir.

- Por ejemplo, cuando se crea una variable u objeto **float** pasando como argumento en el constructor un valor entero:

```
float Real(5);
```

- En este caso este constructor debe realizar internamente la conversión de un tipo de dato entero a otro float.

Destructor:

- En el ejemplo analizado, como se crean los datos del nombre y el apellido de forma dinámica el destructor que crea el compilador automáticamente ya no funciona adecuadamente puesto que sólo destruiría los elementos definidos en la clase y no destruiría las zonas de memoria reservadas dinámicamente por lo que es obligatorio implementarlo:

```
Employee:: ~Employee()
{
  delete[] firstName;
  delete[] lastName;
}
```

CONCLUSIONES