Clase 18. Introducción a la programación orientada a Objetos

Introducción

Conceptos básicos OOP

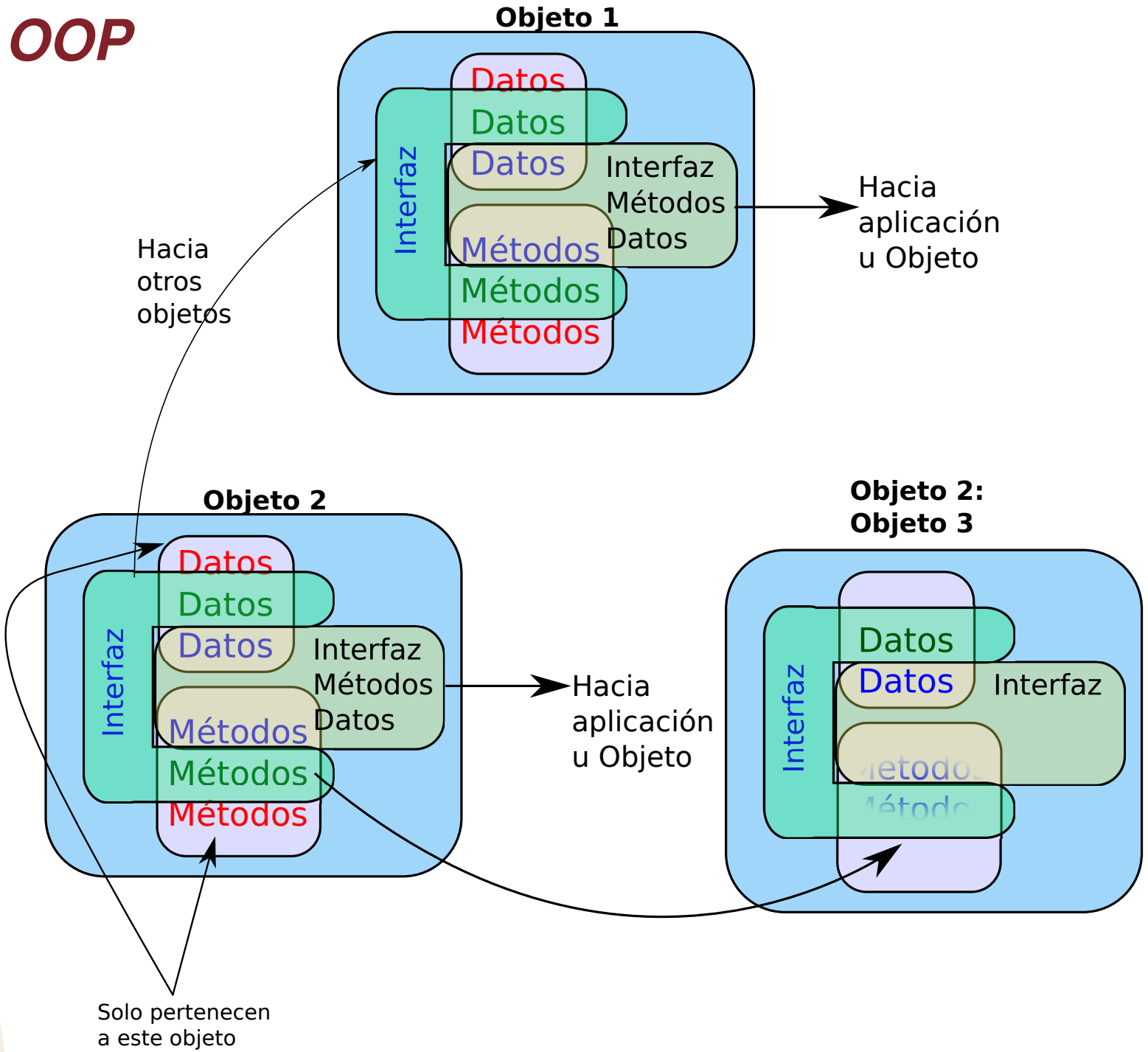
- La programación orientada a objetos es un paradigma de programación en donde los métodos (funciones) y datos (variables) se organizan dentro de entidades de software denominadas *objetos*.

Conceptos básicos OOP

- La programación orientada a objetos es un paradigma de programación en donde los métodos (funciones) y datos (variables) se organizan dentro de entidades de software denominadas *objetos*.
- Un objeto es una instancia (variable) de un tipo de dato abstracto, cuyo comportamiento está definido por métodos y datos asociados al mismo. Además tiene las características de *herencia* y polimorfismo.

Conceptos básicos OOP

- La programación orientada a objetos es un paradigma de programación en donde los métodos (funciones) y datos (variables) se organizan dentro de entidades de software denominadas *objetos*.
- Un objeto es una instancia (variable) de un tipo de dato abstracto, cuyo comportamiento está definido por métodos y datos asociados al mismo. Además tiene las características de *herencia* y polimorfismo.
- Un tipo de dato abstracto, es un tipo con operaciones asociadas, cuya representación está *oculta*. En un tipo de dato abstracto las operaciones pueden no estar definidas hasta que se utiliza un tipo de dato concreto. Es decir, se puede declarar un operador que realiza cierta función, pero no se define *como* se realizará esa función. Esto le da flexibilidad al tipo, de ser utilizado de muchas formas, la forma se puede definir hasta que se determine con que datos de entrada/salida se trabajará.



OOP

- **Objeto** Es la unidad básica de la OOP. Contiene ambos, los datos y funciones que operan sobre los datos.
- **Clase** Es una estructura de datos que define los datos y funciones miembros de un objeto. Un objeto es una *instancia* de la clase.


```
1  #include <iostream>
   using namespace std;
3  class ImprimeMsg{
       public:
5          /*Datos miembro*/
           char *msg="Hola mundo!";
7          /*Metodo miembro*/
           int print();
9  };

11 int ImprimeMsg::print(){
     cout<< "Mensaje: " << msg << endl;
13 }
   int main()
15 {
     ImprimeMsg print; //Instancia
17   print.print();
   return 0;
19 }
```

OOP

- **Método**. Es el comportamiento del objeto, los métodos actuan sobre los datos (son funciones basicamente).
- **Variables de la instancia**. Cada objeto tiene un conjunto *único* de variables.


```
1  #include <iostream>
   using namespace std;
3  class ImprimeMsg{
       public:
5          /*Datos miembro*/
           char *msg="Hola mundo!";
7          /*Metodo miembro*/
           int print();
9  };
   int ImprimeMsg::print(){
11   cout<< "Mensaje: " << msg << endl;
   }
13 int main(){
     ImprimeMsg print; //Instancia
15   print.print();
   return 0;
17 }
```

Conceptos OOP

- **Abstracción.** *Def. 1.* La abstracción consiste en proveer únicamente la información relevante hacia afuera de la clase y esconder los *detalles*.
  Def. 2. Mostrar hacia afuera solo la forma de interactuar con los métodos y datos relevantes, dejando que otros objetos *definan* estos métodos. Por ejemplo, definimos una clase *forma*, y forma tiene una función abstracta *perímetro*, que no está definida en forma, sino que será definida de acuerdo al *tipo* de forma que se instancie: cuadrado, triangulo, circulo, etc.Se puede interactuar con el método perímetro de la clase *forma* sin necesidad de saber como se instancia, o que procedimientos realiza exactamente.

Conceptos OOP

- **Encapsulamiento**. El encapsulamiento consiste en organizar en la misma clase métodos y datos para el mismo proposito. Por ejemplo, una clase *matriz* puede tener un método de solución del sistema por LU, en el mismo objeto estarán las dimensiones, los elementos de la matríz, y las matrices auxiliares L y U, y puede ser que solo los dos primeros sean mostrados hacia afuera del objeto, mientras que L y U son miembros del objeto pero no tienen porque mostrarse al exterior. La idea principal es que todos los elementos necesarios están encapsulados en un solo objeto y que el *cliente* solo necesita conocer algunos de estos elementos.

Conceptos OOP

- **Herencia.** La herencia es basicamente la reutilización del código de una clase (conocida como clase padre o base) para definir una nueva clase (hija o derivada), la clase derivada *hereda* los métodos y datos de la clase base, y además puede definir nuevos.

Conceptos OOP

- **Polimorfismo**. El polimorfismo permite que un método o dato se comporte o instancie diferente, aunque es declarado con el mismo nombre/sintaxis. Por ejemplo, declaro un dato *lista* que puede ser de varios tipos diferentes, el tipo particular se define en tiempo de ejecución. En C++, basicamente el mismo nombre de variable o función puede comportarse o instanciarse diferente de acuerdo a datos de tiempo de ejecución. Declaro un método abstracto en una clase base, pero la defino en la clase heredada.

Conceptos OOP

- **Sobrecarga de funciones**. Una forma particular de polimorfismo es declarar funciones u operadores que se comportan diferente con diferentes datos de entrada/salida, pero que tienen el mismo nombre, por ejemplo, el operador de asignación para un vector, si recibe un vector (del lado derecho) copia cada posición de uno al otro, si recibe un número asigna ese valor a todos los elementos del vector del lado izquierdo. Una particularidad del polimorfismo, es que, usualmente se define el comportamiento de la funcion en tiempo de compilacion, mientras que el polimorfismo en general puede ser en compilación o ejecución.

Resumen

- El **encapsulamiento** es poner todo los elementos necesarios en un solo objeto, se vale del escondimiento de datos **(data hidding)** para mostrar al cliente solo los necesarios para su uso.
- La **abstracción** permite que los métodos y datos puedan ser declarados sin ser definidos. Se sabe que un método abstracto realizará cierta función, pero no es necesario que sepamos como, entonces es posible extender las capacidades del método agregando nuevos comportamientos (al mismo método).
- Cuando un método/dato, abstracto o no, es llamado bajo el mismo nombre pero realiza diferentes funciones, entonces se está haciendo uso del polimorfismo.
- Si por ejemplo el operador $+$ se comporta diferente si lo aplicamos a un número o a una letra, podemos decir que este operador está **sobrecargado**, la sobrecarga solo es una forma de polimorfismo, la otra es cuando un método abstracto realiza varias funciones diferentes (el método abstracto con el mismo nombre).
- En la sobrecarga, los datos de entrada/salida definen el comportamiento de la función, en el otro tipo de polimorfismo, los datos pueden ser los mismos, pero los objetos a los que pertenece el método son diferentes y eso define un comportamiento diferente.

Todos los conceptos están relacionados..

Archivos

En C++ no es necesario utilizar una extensión específica, aunque las mas usuales son: cpp para definiciones, y hpp o h para declaraciones:

print.h

```
#include <iostream>
#ifndef PRINT_H
#define PRINT_H
class ImprimeMsg{
    public:
        /*Datos miembro*/
        char *msg="Hola mundo!";
        /*Metodo miembro*/
        int print();
};
#endif
```

print.cpp

```
#include "print.h"
using namespace std;
int ImprimeMsg::print(){
  cout<< "Mensaje: " <<
    msg << endl;
}
```

```
#include "print.h"
int main(){
  ImprimeMsg print; //Instancia
  print.print();
return 0;
}
```

Compilación

La compilación se realiza igual que la de C, solo hay que cambiar el compilador por g++ (o escoger este tipo de proyecto en codeblocks o en el IDE que utilicen).
Compilación manual:

```
g++ -c print.cpp
g++ -c main.cpp
g++ -o exec print.o main.o
```

Cualquier programa en C es un programa valido de C++.

Tipos de datos básicos y operadores

Los tipos de datos básicos en C++ son lo mismos que en C, así como los operadores. Sin embargo, la definición de una clase se puede ver como la declaración de un nuevo tipo, y los operadores pueden ser *sobrecargados* para comportarse de cierta forma de acuerdo al tipo.

namespace

Un **namespace** es un contexto dentro del cual se definen funciones o datos que solo son validos bajo ese contexto.

namespace

Un **namespace** es un contexto dentro del cual se definen funciones o datos que solo son validos bajo ese contexto.

- Un namespace se declara mediante la directiva: **namespace** y un nombre, el bloque subsecuente define las funciones y variables que pertenecen a ese **namespace**:

```
namespace espacio{
    int x=5;
    int mifuncion(int x, int y);
}
```

namespace

Un **namespace** es un contexto dentro del cual se definen funciones o datos que solo son validos bajo ese contexto.

- Un namespace se declara mediante la directiva: **namespace** y un nombre, el bloque subsecuente define las funciones y variables que pertenecen a ese **namespace**:

```
namespace espacio{
    int x=5;
    int mifuncion(int x, int y);
}
```

- Para indicarle al compilador que el código subsecuente pertenece a otro namespace se utiliza la directiva **using namespace**.

```
using namespace espacio;
x=-12;
```

namespace

Un **namespace** es un contexto dentro del cual se definen funciones o datos que solo son validos bajo ese contexto.

- Un namespace se declara mediante la directiva: **namespace** y un nombre, el bloque subsecuente define las funciones y variables que pertenecen a ese **namespace**:

```
namespace espacio{
    int x=5;
    int mifuncion(int x, int y);
}
```

- Para indicarle al compilador que el código subsecuente pertenece a otro namespace se utiliza la directiva **using namespace**.

```
using namespace espacio;
x=-12;
```

- Las variables y funciones definidas fuera de un **namespace** son globales.

namespace


```
 #include <stdio.h>
namespace espacio{
    int x=5; int mifuncion(int x, int y);
}
namespace espacio2{
    int x=3; int mifuncion(int x, int y);
}
int espacio::mifuncion(int x, int y){
  return (x+y);
}
int espacio2::mifuncion(int x, int y){
  return (x*y);
}
int main(){
  using namespace espacio2;
  x=-12;
  printf("x=%d &x=%p &x=%p\n",x,&x,&espacio2::x);
  espacio::x=-5;
  printf("x=%d \n",x);
return 0;
}

```

Entrada y salida estándar

La entrada y salida estándar en C++ están definidas por cin y cout respectivamente, definidas en ¡iostream¿ dentro del **namespace std**. (También ahí están cerr y clog, para la salida estándar de logs y errores).

```
1  cout << "Hola mundo\n"; //Imprime caracteres
   cout << 12.54; //Imprime un numero
3  cout << x; //Imprime el valor de una variable
   cout<< "Hola" << 12.56 << x << endl;
5  cin>> x; //Lee la variable x
   cin >> x >> y; //Lee las variables x y y
7  cout<< setprecision(5) << x
```

Se puede modificar el formato de la salida (para flotantes) con **setprecision**. **Puede notar que el operador << está sobrecargado para el tipo de dato que reciba**.

Clases

Una **clase** es la definición para un tipo de dato. La clase no define el dato, sino define el significado del nombre de la clase, define que datos y métodos contiene una *instancia* de la clase. a la definición de la clase se le antepone la palabra reservarda **class**.

```
1 class ClasePrueba
{
3   public: //visibilidad de los datos
      double x, y, z;
5 };
```

Para requerir memoria estática una *instancia de la clase* se antepone el nombre de la clase a la instancia:

```
1 ClasePrueba Instancia;
```

Clases

De forma genérica:

```
1 class NombreClase
  {
3   especificador_de_acceso:
                  tipo miembro;
5                 tipo funcion_miembro();
    especificador_de_acceso:
7                 tipo miembro2;
                  tipo funcion_miembro();
9 };
```

Clases: definición funciones miembro


```
1  class Prueba
   {
3    public:
           double x,y;
5          double suma();

7  };

9  double Prueba::suma()
   {
11   return (x+y);
   }
13
```

Clases: acceso a miembros


```
#include <iostream>
using namespace std;// Para cout
class Prueba{
  public:
        double x,y;
        double suma();
};
double Prueba::suma(){
  return (x+y);
}
int main()
{
  Prueba X;
  X.x=2.3; X.y=48;
  cout<< "Suma: "<< X.suma() << endl;
}
```

Memoria dinámica

Por lo pronto veremos solo memoria dinámica para tipos básicos (int, float, double, etc.) después lo extenderemos a clases.

- Los operadores **new** y **delete** sirven para requerir y devolver memoria dinámica respectivamente. Ej:

```
int *ptr=new int[10]; //memoria para 10 enteros
delete []ptr; //Devuelve "bloque" de memoria
```

Tarea 10

prog10.1 Defina una clase cuyos miembros **publicos** sean:

1. 2 apuntadores a double y un entero **n**.
2. Una función lee, que leé de la entrada estándar con *cin* el valor entero **n** y $2n$ valores que se insertarán en los vectores.
3. Una función que pide memoria para los dos vectores y una que la devuelve con **new** y **delete**.
4. Una función que hace la suma de los dos vectores e imprime a pantalla el resultado(no recibe parámetros), otra el promedio (de los dos, tampoco recibe parámetros devuelve un double), otra el producto punto.
5. La definición de la clase (donde está **class** con el nombre y los miembros, se hará en un .h, la implementación de las funciones en un .cpp, el main en otro .cpp, se compilan por separado y se ligan.

Nota: si se usa el: **using namespace** dentro de una función su alcance se restringe al bloque de la función (lo cual es deseable), se requiere para cout y cin.