Práctico de Programación 4 – clase 1

Práctico 1, Ejercicio 4:

Se pide implementar en C++ una clase que represente un punto en un plano, utilizando coordenadas cartesianas.

En principio, una clase de C++ es similar a un estructurado de C que además permite incluir operaciones, pero esa no es la única diferencia.

```
class Punto {
   private:
      float x, y;
};
```

Un objeto de clase `Punto` (terminología de Orientación a Objetos análoga a decir "una variable de tipo `Punto`") tiene dos miembros (análogo a "campo" o "atributo") de tipo flotante para representar sus coordenadas. La etiqueta `private:` es una palabra reservada e indica que no podré acceder desde una operación que no pertenece a la clase `Punto`, a todos los miembros que se declaren a continuación (hasta el final de la declaración de la clase o hasta encontrar una etiqueta distinta).

En particular, si declaro un objeto `Punto` de la siguiente forma:

```
Punto p;
```

no podré hacer lo siguiente (se obtiene un error de compilación):

```
p.x = 3.2;
cout << p.x;
```

Para poder hacer lo anterior, los miembros `x` e `y` deberían estar bajo una etiqueta `public:` pero justamente la idea es utilizar el mecanismo de *ocultamiento de información* que proporciona el lenguaje. Por lo tanto, para acceder a los miembros privados de la clase `Punto`, necesitamos declarar e implementar operaciones públicas, de la siguiente forma:

```
// Declaración
class Punto {
   private:
      float x, y;
   public:
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};

// Implementación
void Punto::setX(float x) {
    this->x = x;
}

float Punto::getX() {
    return this->x:
}
```

Observaciones sobre las implementaciones de las dos operaciones anteriores:

- La clase `Punto` *encapsula* datos y operaciones en una misma declaración.
- El prefijo `Punto::` que antecede al nombre la operación cuando esta se implementa, es para indicar que la misma pertenece (es miembro) de la clase.
- La palabra reservada `this` representa un puntero al *objeto implícito*, es decir, al objeto a través del cual se invoca a la operación. En el caso de la operación `setX` es necesario utilizarlo para evitar ambigüedad, pues en la asignación `x = x` el compilador no tiene elementos para distinguir al miembro privado de la clase, del parámetro de entrada. Una alternativa sería utilizar otro nombre para el parámetro, por ejemplo `unX`, y luego hacer `x = unX`; en este caso no sería necesario acceder a través de `this`, pues se asume que el identificador `x` refiere a un miembro de la clase. Algunos estilos de codificación en C++ recomiendan siempre utilizar `this` para evitar ambigüedades; como contraparte, el código resulta más cargado de caracteres.

Notar que las operaciones no reciben como parámetro al punto, dado que se trabaja con el objeto implícito. Una forma más clara de ver esto, es a través del siguiente ejemplo:

```
Punto p;

p.setX(3.2);
cout << p.getX();
```

En ambos casos, las operaciones se invocan a través del objeto. En el contexto de las invocaciones a `setX` y `getX`, el objeto implícito será `p`.

Por otra parte, si quisiéramos inicializar puntos de la siguiente forma:

```
Punto p1, p2(3.0, 4.5); // Quiero que p1 se inicialice con las coordenadas
                        // (0.0, 0.0) por defecto y que p2 se inicialice
                        // con los valores pasados como parámetros en x e y
                        // respectivamente
```

Hay que implementar explícitamente el constructor por defecto y el constructor por parámetros:

```
class Punto {
   private:
      float x, y;
   public:
      Punto();
      Punto(float, float);
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};

Punto::Punto() {
    this->x = 0.0;
    this->y = 0.0;
}

Punto::Punto(float x, float y) {
    this->x = x;
    this->y = y;
}
```

O alternativamente, las dos operaciones en una misma:

```
class Punto {
   private:
      float x, y;
   public:
      Punto(float =0.0, float =0.0);
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};
```

De esta forma no se implementa el constructor por defecto y el código del constructor por parámetros es exactamente el mismo al ya implementado.

Observaciones:

- Notar que los constructores tienen una sintaxis particular. No retornan nada pero no se declara un resultado `void`.
- Estos constructores se invocan automáticamente cuando se crea un objeto de la clase correspondiente. La memoria ya está reservada y los constructores solamente realizan acciones de inicialización.

Por otro lado, notar que sin implementar nada más es posible hacer lo siguiente, con el comportamiento habitual:

```
{  // Comienzo del bloque, se asigna automáticamente memoria estática
   // parap1 y p2

  Punto p1, p2(3.0, 4.5);
  p1 = p2; // Asigna el contenido de p2 en p1

}  // Fin del bloque, se libera automáticamente la memoria estática ocupada
   // por p1 y p2
```

Cuando una clase maneja explícitamente memoria dinámica (a través de `new` y `delete`), es necesario implementar explícitamente otros constructores, el operador de asignación y el destructor. Se verá en la clase 2.

De la misma forma que se pueden declarar arreglos de variables, se pueden declarar arreglos de objetos, por ejemplo:

- Arreglo estático:

```
Punto pts[10];
// Crea un arreglo de 10 objetos de clase Punto e inicializa a cada
// uno invocando al constructor por defecto.
// Al salir del bloque {} donde se declaró pts, se libera la memoria
// estática ocupada por los 10 objetos.
```

- Arreglo dinámico:

```
Punto *pts;
pts = new Punto[10];
```

```
      // Obtiene memoria dinámica para 10 objetos de clase Punto e
      // inicializa a cada uno invocando al constructor por defecto.

      delete [] pts;
      // Libera la memoria dinámica ocupada por los 10 objetos.
```

En este ejercicio, se pide además implementar una clase que represente un segmento en un plano, que incluya una operación que calcule el largo del segmento.

```
class Segmento {
   private:
      Punto p1, p2;
   public:
      Segmento();
      Segmento(Punto, Punto);
      Segmento(Punto);
      Segmento(float, float, float, float);
      float largo();
};
```

Observaciones:

- Una clase (en este caso, `Segmento`) puede tener como miembros a objetos de otras clases (`Punto`), de la misma forma que los tipos elementales (como `Punto` tiene miembros de tipo `float`).
- Pueden existir diferentes constructores por parámetros según se necesite. En este caso hay tres: (1) para crear un segmento a partir de sus dos puntos extremos, (2) para crear un segmento a partir de un punto y el origen (0.0, 0.0), y (3) para crear un segmento a partir de las coordenadas de sus dos puntos extremos.

Implementación de la clase:

```
Segmento::Segmento() {
}
// Se inicializan los miembros p1 y p2 con el constructor por defecto de
// Punto

Segmento::Segmento(Punto p1, Punto p2) {
    this->p1 = p1;
    this->p2 = p2;
}
```

Alternativamente (y más eficiente), el constructor anterior se puede implementar de la siguiente forma:

```
Segmento::Segmento(Punto unP1, Punto unP2) : p1(unP1), p2 (unP2){
}
// El código después de los : indica la inicialización de los miembros
```

Las demás operaciones:

```
Segmento::Segmento(Punto p) : p1(p) {
}
// Se inicializa p1 con el parámetro p y p2 con el valor por defecto
// (0.0, 0.0)
```

```
Segmento::Segmento(float x1, float y1, float x2, float y2) :
                   p1(x1, y1), p2(x2, y2) {
}
// Inicializa los miembros p1 y p2 del segmento, invocando al contructor
// por parámetros de Punto
```

Notar que lo siguiente no sería válido (no compila):

```
Segmento::Segmento(float x1, float y1, float x2, float y2) {
    this->p1.x1 = x1;
    this->p1.y1 = y1;
    this->p2.x2 = x2;
    this->p2.y2 = y2;
}
```

dado que las coordenadas son miembros privados de `Punto` y por lo tanto no se puede acceder a ellas desde `Segmento`. Alternativamente se podría implementar de la siguiente forma, pero no es práctico:

```
Segmento::Segmento(float x1, float y1, float x2, float y2) {
    this->p1.setX(x1);
    this->p1.setY(y1);
    this->p2.setX(x2);
    this->p2.setY(y2);
}
```

Las demás operaciones:

```
float Segmento::largo() {
    return sqrt(pow(p2.getX() - p1.getX(), 2) +
                pow(p2.getY() – p2.getY(), 2));
}
```

Para utilizar las funciones `pow` y `sqrt` se debe incluir la biblioteca `cmath`.

Organización de los archivos fuente:

- Usualmente se separa la declaración de una clase (archivo `.h`), de su implementación (archivo `.cpp`).
- Lo anterior permite ocultar los detalles de implementación de las operaciones, si bien no se oculta la estructura interna de la clase. Notar que el cliente de la clase (otra clase que necesite utilizarla), basta con que conozca el archivo `.h` (el cual tiene que incluir, mediante la directiva al precompilador `#include`) y el archivo `.cpp` compilado (código objeto).
- Entonces por cada clase habrá un archivo `.h` y otro `.cpp`. No se recomienda declarar varias clases dentro de un mismo archivo `.h`, dado que quedarían acopladas. Esta recomendación se flexibiliza cuando se trata de varias clases altamente acopladas, por ejemplo, que forman parte de una misma biblioteca.
- Dado que una misma clase puede necesitarse desde diferentes clientes (`#include` de un mismo archivo desde varios archivos diferentes), debe evitarse que el compilador reciba varias copias del mismo código, lo que causaría un error de compilación debido a múltiples

definiciones de un mismo identificador. Para eso se utilizan las directivas al precompilador `#ifndef, #define, #endif`. A continuación, se muestra un ejemplo completo:

Archivo Segmento.h

```
#ifndef SEGMENTO
#define SEGMENTO

#include "Punto.h"

class Segmento {
   private:
      Punto p1, p2;
   public:
      Segmento();
      Segmento(Punto, Punto);
      Segmento(Punto);
      Segmento(float, float,
               float, float);
      float largo();
};

#endif
```

Archivo Segmento.cpp

```
#include "Segmento.h"
#include <cmath>

Segmento::Segmento() {
}

Segmento::Segmento(Punto unP1,
                   Punto unP2) :
          p1(unP1), p2 (unP2){
}

Segmento::Segmento(Punto p) : p1(p) {
}

Segmento::Segmento(float x1, float y1,
                   float x2, float y2) :
          p1(x1, y1), p2(x2, y2) {
}

float Segmento::largo() {
    return sqrt(
           pow(p2.getX()-p1.getX(), 2) +
           pow(p2.getY()-p2.getY(), 2));
}
```

Observaciones sobre el código anterior:

- Por convención se nombra a los archivos `.h` y `.cpp` con el mismo identificador de la clase.
- La directiva `#ifndef` causa que cuando el precompilador procese el archivo `Segmento.h`, pregunte si en su tabla de símbolos está definido el identificador `SEGMENTO` (como convención se usa el mismo nombre de la clase, pero todo en mayúscula). Si no está definido, entonces lo ingresa en dicha tabla (mediante `#define`) y procesa todo el código. Si ya está definido (porque ya se incluyó desde otra clase cliente), entonces no se procesa. Notar que la primera línea debe contener la directiva `#ifndef`, mientras que la última debe indicar `#endif`.
- El archivo `Segmento.h` debe incluir al archivo `Punto.h`, dado que la clase `Segmento` tiene miembros de clase `Punto`. El texto entre comillas del `#include` denota el camino hasta el archivo, puede incluir directorios en caso de que el archivo `Punto.h` esté en otro directorio diferente al actual.
- En `Segmento.cpp` se debe incluir `Segmento.h`, dado que se necesita la declaración para compilar la implementación. Además, se incluye `cmath` (en este caso entre llaves <>, por ser una biblioteca estándar), dado que se necesita para la implementación de la operación `largo`.

Práctico 1, Ejercicio 5:

Se pide implementar una clase que permita representar fechas, con día, mes y año.

La primera decisión que se debe tomar refiere a la representación interna, para la cual en principio se pueden manejar dos opciones:

1. Tres miembros de tipo entero para el día, mes y año, respectivamente.
2. Un entero largo que represente la cantidad de días desde una fecha inicial.

Ambas opciones tienen ventajas y desventajas. La opción 1 permite mostrar directamente la fecha sin realizar conversiones, mientras que algunas operaciones pueden tener una lógica compleja. La opción 2 facilita algunas operaciones, pero requiere realizar una transformación para su visualización, además de restringir el rango de fechas que se puede manejar (depende de la fecha inicial).

Para esta resolución se tomará la opción 1, sin embargo, es importante tener claro que una representación interna intuitiva no necesariamente es la más adecuada en determinadas situaciones. Por simplicidad, asumiremos que todos los meses tienen 30 días.

```
class Fecha {
  private:
    int dia, mes, anio;
  public:
    Fecha(int, int, int);
    Fecha avanzar(int);
    Fecha retroceder(int);
    int diferencia(Fecha);
    bool igual(Fecha);
};

Fecha::Fecha(int dia, int mes, int anio) {
    this->dia = dia;
    this->mes = mes;
    this->anio = anio;
}

Fecha Fecha::avanzar(int inc) {
     int nDia = (this->dia + inc) % 30;
     int nMes = (this->mes + (this->dia + inc) / 30)) % 12;
     int nAnio = (this->anio) + (this->mes + (this->dia + inc) / 30) / 12;

     return Fecha(nDia, nMes, nAnio); // Devuelve una nueva fecha
}

// La implementación de la operación retroceder as análoga a la de avanzar

int Fecha::diferencia(Fecha f) {
     long int f1 = f.dia + f.mes*30 + f.anio*12*30;
     lont int f2 = this->dia + this->mes*30 + this->anio*12*30;

     return f2 - f1;
}

bool Fecha::igual(Fecha f) {
     return this->dia==f.dia && this->mes==f.mes && this->anio==f.anio;
}
```

Ejemplo de declaración de objetos e invocación a operaciones:

```
Fecha f1(12, 1, 2020), f2(14, 1, 2020), f3;
int dif;

f3 = f2.avanzar(4);
dif = f2.diferencia(f1);
if (f1.igual(f2)) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Notar que sería más cómodo poder hacer lo siguiente:

```
f3 = f2 + 4;
dif = f2 - f1;
if (f1 == f2) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Para lograr lo anterior, hay que hacer los siguientes cambios, tanto en la declaración como en la implementación de la clase Fecha:

- Sustituir `avanzar` por `operator+`
- Sustituir `diferencia` por `operator-`
- Sustituir `igual` por `operator==`

Con lo anterior, estamos haciendo uso del mecanismo de sobrecarga de operadores de C++. El código anterior es una forma abreviada de lo siguiente (que también compila, pero claramente no es cómodo de usar):

```
f3 = f2.operator+(4);
dif = f2.operator-(f1);
if (f1.operator==(f2)) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Al momento de sobrecargar operadores, tener en cuenta lo siguiente:

- Hay un conjunto predefinido de operadores que se pueden sobrecargar. Por ejemplo, para `Fecha` podría ser útil sobrecargar el `operator++`:

```
void Fecha::operator++() {
    *this = *this + 1;
}
// Modifica el objeto implícito, invocando a la operación que
// incrementa una fecha, pasando como parámetro el valor de un día.
```

- La cantidad y tipo de los parámetros no están predefinidos, tampoco están relacionados con el operador que se está sobrecargando. Esto significa que hay relativa libertad al momento de sobrecargar los operadores.
- Debido al punto anterior, se debe cuidar de no sobrecargar un operador con un comportamiento que no es el esperado. Por ejemplo, en el siguiente código:

  ```
  String s1, s2, s3;
  s3 = s1 + s2;
  ```

  es razonable esperar que en `s3` se asigne la concatenación de `s1` y `s2`. Pero nada impide que yo sobrecargue el `operator+` en `Fecha` para que decremente su valor, cosa que no es deseable.

- Un mismo operador se puede sobrecargar para distintas clases (por ejemplo, `operator+` está sobrecargado para `String` y también para `Fecha`), pero también se puede sobrecargar en una misma clase, siempre que tenga diferentes parámetros, por ejemplo, en `Fecha` podemos declarar las siguientes operaciones:

  ```
  Fecha operator-(int);
  // Decrementa a la fecha implícita en una cantidad de días indicada
  // por el parámetro

  int operator-(Fecha);
  // Calcula la diferencia en días entre dos fechas
  ```

  Tener en cuenta que sumar dos fechas en principio no tiene sentido, por más que sintácticamente el lenguaje lo permita.

Finalmente, notar que no es necesario implementar el operador de asignación ni el destructor en la clase `Fecha`, dado que el comportamiento por defecto es el deseado.

La implementación del operador de inserción de flujo (`<<`) se verá en la siguiente clase de práctico.