Índice

Que es la POO

La POO es un paradigma de programación que usa objetos en sus interacciones. Es cercano a como expresamos las cosas en la vida real en nuestro día a día. En la POO en lugar de determinar qué funciones necesitamos (como hacíamos en la programación estructurada) se debe determinar qué **objetos** se necesitarán. Los problemas se irán resolviendo de abajo hacia arriba, desde pequeños objetos que actuarán entre sí y conformarán otros más complejos.

Hay 3 enfoques en la Programación Orientada a Objetos: diseño (TDA), implementación (Clase), usuario (Objeto). En general, se trabaja de manera grupal y hay un grupo para cada uno.

TDA

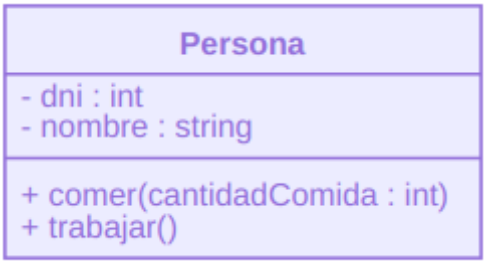
El TDA (Tipo de Dato Abstracto) es un concepto matemático, un mecanismo de descripción de alto nivel que al implementarse genera una clase. El TDA es independiente de la implementación y del lenguaje. Vendría a ser un plano del código, y generalmente se hace con un diagrama UML

UML

UML (Lenguaje de Modelado Unificado) es un lenguaje de modelado, **no** de programación. Cuando un código incluye muchas clases debe tener un UML, ya que sirve como documentación.

Existen 3 modificadores de acceso que indican quienes pueden acceder a un atributo o método

- Publico (+) : **todos** pueden acceder, tanto fuera como dentro de la clase.
- Protegido (#) : solo **la clase y sus hijos** pueden acceder
- Privado (-) : **solo la clase** puede acceder



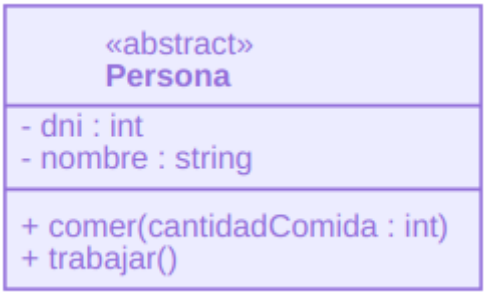
Observación: en general los getters y setters no se incluyen en el diagrama, pero se asume que están en la implementación.

Clases y métodos abstractos

Este tema se verá más adelante, pero básicamente un método abstracto es aquel que se encuentra declarado en la clase pero no implementado, debe tener en su declaración la palabra clave **virtual**.

Una clase abstracta es aquella que tiene **al menos un método abstracto**, esta clase no puede ser instanciada.

Hay dos formas de escribir una clase abstracta en UML, una es:



La otra es con 'Persona' en itálica: *Persona*.

Clase

La clase es la implementación de un TDA, en este punto sí se piensa un lenguaje, en cómo se va a hacer, etc.

Un ejemplo de clase implementada en C++:

```
// Persona.h
class Persona {
    // Atributos
    private:
        int dni;
        string nombre;
    // Metodos
    public:
        // Constructor con parametros
        Persona(int dniOut, string nombreOut);
        // getters
        Persona obtenerDni();
        Persona obtenerNombre();
        // setters
        Persona asignarDni();
        Persona asignarNombre();
        void comer(int cantidadComida);
        void trabajar();
};

// Persona.cpp
Persona:: Persona (int dniOut = 0, string nombreOut = " ") {
    dni = dniOut;
    nombre = nombreOut;
}

Persona Persona:: obtenerDni() { return dni; }
Persona Persona:: asignarDni(int dniOut) { dni = dniOut; }
Persona Persona:: obtenerNombre() { return nombre; }
Persona Persona:: asignarNombre(string nombreOut) { nombre = nombreOut; }

void Persona:: comer(int cantidadComida) {
    cout << nombre << " esta comiendo " << cantidadComida << "gr de comida";
}
void Persona:: trabajar() {
    cout << nombre << " esta trabajando";
}
```

Observaciones

- Los atributos son privados, para que no se pueda acceder a ellos desde "afuera"
- Para instanciar una clase hay que hacer un constructor (sino el lenguaje asigna uno de oficio, que por defecto inicializa todos los atributos en cero)
- Los **constructores** son métodos especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara. Por convención, tienen el mismo nombre que la clase, no retornan nada, no pueden ser heredados y deben ser públicos.
- Los **destructores** son funciones con el mismo nombre que la clase pero precedidos por un ~. Se utilizan cuando se trabaja con memoria dinámica y **NO** pueden sobrecargarse. **Observación**: los destructores se llaman automáticamente al finalizar la función/método, no los llamamos nosotros

Objeto

El objeto es una clase instanciada, por ejemplo si tenemos la clase Persona, podemos instanciar el objeto Valentina

Atributos públicos:

```
const int GR_ALMUERZO = 600
int main() {
     Persona valentina; // Objeto
     valentina.dni = 43424140;
     valentina.nombre = "Valentina";
     valentina.comer(GR_ALMUERZO);
     valentina.trabajar();
     ...
 }
```

El problema con lo anterior es que se podrían hacer modificaciones a los atributos desde cualquier lugar, y no es la idea. Por eso es que los atributos son **privados**, y si queremos inicializar el objeto lo hacemos con un constructor. La forma correcta de instanciar un objeto seria con el constructor:

```
const int GR_ALMUERZO = 600
int main() {
     Persona valentina(43424140, "Valentina");
     valentina.comer(GR_ALMUERZO);
     valentina.trabajar();
    ...
 }
```

Pero ¿qué pasa si a la hora de instanciar el objeto no sé cuáles van a ser sus atributos? ¿Y si quiero modificarlo más adelante? ¿O si necesito alguno de los datos? Para esto se utilizan los getters y setters:

```
const int GR_ALMUERZO = 600
int main() {
     Persona valentina(); // Instancio el objeto, pero no lo incializo con
ningun valor
     valentina.asignarDni(43424140); // setter dni
     valentina.asignarNombre("Valentina"); // setter nombre
     valentina.comer(GR_ALMUERZO);
     valentina.trabajar();
    ...
 }
```

Sobrecarga de métodos

Sobrecargar un método, es tener dos o más métodos con el mismo nombre siempre y cuando **difieran** en

- La cantidad de parámetros
- Los tipos de los parámetros
- Ambas cosas

Los constructores también pueden sobrecargarse. Volviendo a la clase Persona, en vez de declarar un constructor con parámetros que tenga valores por defecto, podría hacer un nuevo constructor sin parámetros.

Hay dos formas de escribir eso, la primera y probablemente la más intuitiva es esta

```
// Persona.cpp
Persona:: Persona () {
    dni = 0;
    nombre = " ";
}
```

La otra es

```
// Persona.cpp
Persona:: Persona (): dni(0), nombre(" ") {}
```

Personalmente prefiero la segunda si no hay muchos atributos.

Sobrecarga de operadores

La sobrecarga de operadores es similar a la de métodos, pero aplica para los operadores. Ahora bien, ¿por qué querría sobrecargar un operador? Supongamos que tengo la clase `Fracción` y quiero sumar dos fracciones

```
// Fraccion.h
class Fraccion {
    private:
        int numerador, denominador;
        int maximoComunDivisor(int n, int d);
    public:
        Fraccion(int n = 0, int d = 1); // Constructor con parámetros
};
// Fraccion.cpp
Fraccion:: Fraccion (int n, int d){
    numerador = n;
    if (d == 0) {
        cout << endl << "ERROR: intento de division por cero" << endl;
        exit(0);
    } else denominador = d;
}

int Fraccion:: maximoComunDivisor(int n, int d) {
    while (n != d) {
        if (n > d) n -= d;
        else d -= n;
    }
    return n;
}
// main.cpp
 int main()
    {
        Fraccion a(2,3), b(1,3);
        Fraccion resultado;
        resultado = a + b; // ERROR
```

```
        return 0;
    }
```

El compilador no sabe qué hacer cuando llega a `Fraccion resultado = a + b;` El error se debe a que `Fraccion` es en realidad una clase y no un tipo primitivo (como `int`, `void`, etc.) y, en consecuencia, se debe de adiestrar al compilador para que éste sepa de qué manera hará la suma de dos objetos del tipo Fraccion. Para "adiestrar" el compilador, se sobrecarga el operador +

Sintaxis: `tipo  operator + (lista de parámetros);`

- `tipo` se refiere al tipo regresado por el operador
- `operator` es una palabra reservada que debe aparecer en toda declaración de sobrecarga de operadores
- el símbolo `+` está en representación de cualquier operador
- `lista de parámetros` indica los argumentos sobre los que operará la función de sobrecarga

Sobrecargando el operador + :

```
// En Fraccion.h agregar la definición del método
// Fraccion.cpp
Fraccion Fraccion:: operator + (Fraccion otraFraccion) {
    int nuevoNumerador = numerador * otraFraccion.denominador + denominador *
otraFraccion.numerador;
    int nuevoDenominador = denominador * otraFraccion.denominador;
    return Fraccion(nuevoNumerador, nuevoDenominador);
}
```

De esta manera sí se podría hacer `Fraccion resultado = a + b;`

Observaciones

- En C++ hay operadores que **NO** se pueden sobrecargar:
    1. Operadores de directivas de procesador #, ##
    2. Selector directo de componente .
    3. Operador para valores por defecto de funciones de clase :
    4. Operador de acceso a ámbito ::
    5. Operador de indirección de puntero-a-miembro .*
    6. Condicional ternario ?
    7. sizeof
    8. typeid
- Existen dos tipos de operadores: binarios y unitarios.
    - Los operadores **binarios** son aquellos que poseen dos partes (izquierda y derecha), por ejemplo, una operación de suma requiere dos operandos (o1 + o2).
    - Los operadores **unitarios** son aquellos que poseen solo una parte, por ejemplo, una operación de incremento (o1 ++).

Puntero this

Todo objeto se crea con una referencia o puntero a sí mismo, este puntero se llama `this` y se genera de manera automática. Ahora bien, ¿cuándo o por qué se usa? Se *puede* utilizar siempre pero nos veremos ***obligados*** a utilizarlo cuando el compilador no pueda resolver una ambigüedad en los nombres.

Algunos ejemplos:

```
// No se usa this
Persona:: Persona(int dniOut, string nombreOut) {
    dni = dniOut;
    nombre = nombreOut;
}
// Se usa this, pero no es encesario
Persona:: Persona(int dniOut, string nombreOut) {
    this->dni = dniOut;
    this->nombre = nombreOut;
}
// Se usa this y es necesario porque tanto los parámetros como los atributos
tienen el mismo nombre
Persona:: Persona(int dni, string nombre) {
    this->dni = dni;
    this->nombre = nombre;
}
```

Constructor de copia

El constructor de copia es simplemente un constructor más, que si no es programado, es provisto por el lenguaje (constructor de copia de oficio). ¿Cuándo se llama? Cuando se pasa un objeto por parámetro en alguna función o, como cuando se crea un objeto igualándolo a otro, por ejemplo.

Volviendo al ejemplo de la clase Fraccion, las siguientes líneas hacen exactamente lo mismo:

```
1. Fraccion resultado = a + b;
2. Fraccion resultado(a + b);
```

Ambas llaman al constructor copia, ¿por qué? Primero se crea el objeto `a + b`, luego se lo pasa por valor al constructor, por lo que en el constructor se hace una copia de `a + b` para poder luego asignárselo a `resultado`. Básicamente le estoy pidiendo al compilador que construya `resultado` *en base* al objeto resultante de sumar `a` y `b`.

Una forma de evitar llamar al constructor copia seria hacer lo siguiente:

```
Fraccion resultado; // El objeto se instancia con los valores por defecto, n = 0
y d = 1
resultado = a + b;  // Al objeto se le asigna el resultado de a + b
```

Observación 1: el constructor copia de oficio, simplemente se encarga de copiar todos los valores de los atributos de un objeto a otro.

El parámetro del constructor copia debe ser de la misma clase que estamos programando, dado que se va a copiar de él. Pero **NO** puede pasarse por valor, porque sino se copiaría el mismo parámetro, y se crearía un loop infinito imposible de resolver. Ergo, lo pasamos por referencia.

Ejemplo de como declarar un constructor copia:

```
Fraccion:: Fraccion(const Fraccion& otraFraccion) {
    numerador = otraFraccion.numerador;
    denominador = otraFraccion.denominador;
}
```

Observación 2: por convención, se suele indicar que esa referencia es constante para "asegurarle al compilador" que el objeto no será modificado

Ahora bien, supongamos que tenemos una clase Vector que trabaja con **memoria dinámica**

```
// Vector.h
class Vector {
    private:
        int tamanio;
        int* datos;
    public:
        Vector();
        Vector(int tamanioOut);
        Vector insertarValor(int valor, unsigned pos);
        // Otros métodos
        ~Vector();
};

// Vector.cpp
Vector:: Vector() : tamanio(0), datos(0) {}
Vector:: Vector(int tam) {
    tamanio = tam;
    // Se solicita memoria en el heap para guardar los datos
    datos = new int[tamanio];
}
void Vector:: insertarValor(int valor, unsigned pos) {
    datos[pos-1] = n;
}
Vector:: ~Vector() {
    // Se libera la memoria solicitada
    if (tamanio > 0) delete []datos;
}
```

Si creamos un objeto del tipo Vector, para guardar números del 1 al 5, deberíamos hacerlo de la siguiente manera

```
const int TAMANIO_VECTOR = 5;
Vector vector1(TAMANIO_VECTOR);
for (int i = 0; i < TAMANIO_VECTOR; i++) {
    vector1.insertarValor(i+1, i); // guardo 1 en pos 0, 2 en pos 1, 3 en pos 2,
etc.
}
```

Luego de esto, tendríamos nuestro objeto `vector1` cuyos atributos serán

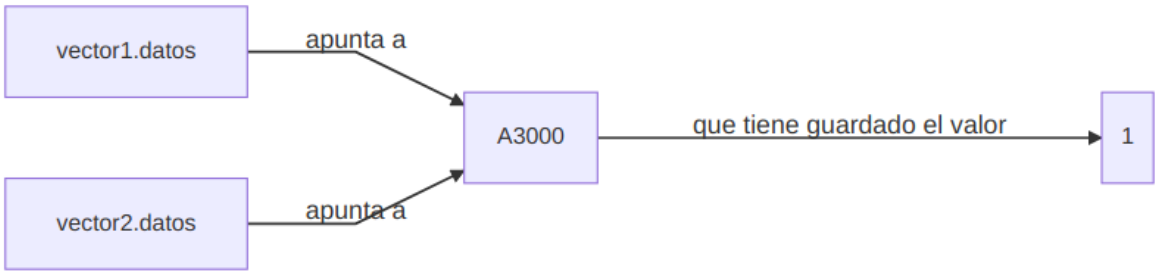
- `tamanio = 5`
- `datos = A3000` (por ejemplo, lo importante es que va a ser la dirección inicial del vector)

Los números 2, 3, 4 y 5 estarán en las celdas contiguas

Nota: No me refiero a celdas estrictamente contiguas, sino a la siguiente celda que apunte a ese tipo de dato, que será la que diste una vez el tamaño del registro de la primera. En el caso de un entero (que ocupa 2 bytes), la siguiente celda será en la que haya un valor será la A3002, luego la A3004 y así sucesivamente.

Observación 3: los valores del vector están guardados en el heap, pero **no** en el objeto vector1. Habría que hacer un método que se encargue de eso. El `vector1` solo guarda el puntero a la primera dirección de memoria que almacena un entero.

Ahora supongamos que quiero instanciar un nuevo vector, y copiarle los valores del `vector1`. Si hago lo siguiente manera `Vector vector2 = vector1;` voy a estar copiando los **atributos** del `vector1` al `vector2`, porque se esta utilizando el constructor copia de oficio. Esto quiere decir que el tamaño de ambos va a ser 5 y que el puntero datos de ambos van a apuntar a la misma dirección de memoria.



Esto no solo no es lo que queremos, sino que además nos genera dos **errores**:

1. Cuando se llame al destructor de `vector1`, se va a liberar la memoria solicitada (que comienza en A3000 y termina 4 posiciones mas adelante) y `vector2` no podrá acceder a los datos.
2. Cuando se llame al destructor de `vector2`, va a intentar liberar memoria que ya había sido liberada

Conclusión: Siempre que se manejen atributos que utilizan memoria dinámica va a ser necesario definir un constructor de copia. Ejemplo de como hacerlo:

```
Vector:: Vector(const Vector &vec) {
    // Los tamaños deben ser iguales
    tamanio = vec.tamanio;
    // Se solicita nueva memoria para guardar los datos
    datos = new int[tamanio];
    // Se copian los valores a esa nueva porcion de memoria
    for (int i = 0; i < tamanio; i++)
        datos[i] = vec.datos[i];
}
```

Características principales de la POO

Encapsulamiento

Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior. Consiste en agrupar en una Clase los atributos con un acceso privado y los métodos con un acceso público. La forma de acceder o modificar los atributos de una clase es a través de sus métodos.

Ahora bien ¿por qué hay que hacer esto? Porque cuando no hay encapsulamiento los atributos pueden tomar valores inconsistentes, lo que seria fatal para cualquier sistema. El encapsulamiento nos ayuda a proteger la integridad de los datos y nos asegura que los atributos de nuestra clase solo podrán ser accedidos a través de los métodos definidos en dicha clase.

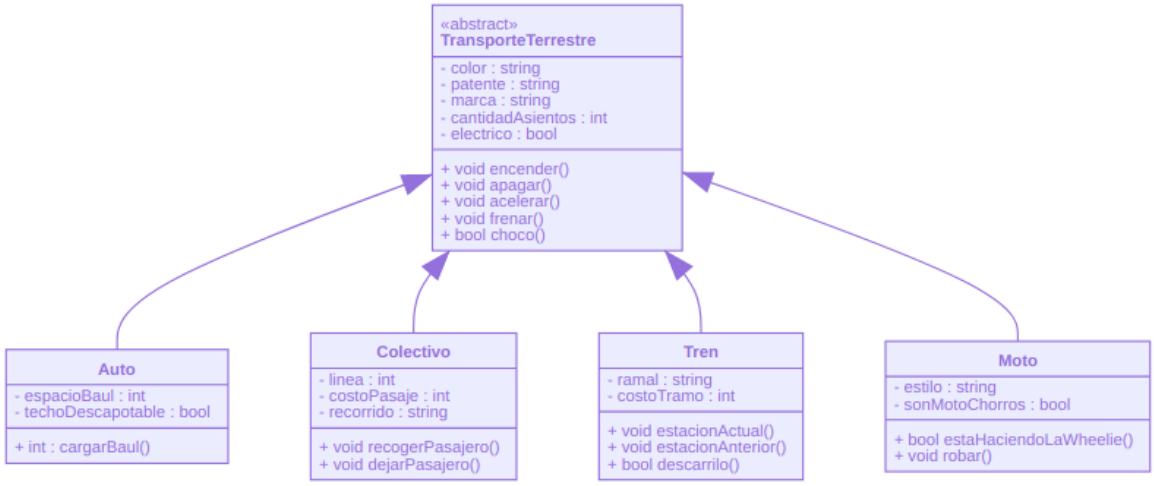
Ocultamiento de la implementación

Con la finalidad de proteger al objeto y a su usuario, los datos deberían ser, en general, inaccesibles desde el exterior para que no puedan ser modificados sin autorización. Para trabajar con ellos se utilizan métodos estipulados para tal fin (en general se los llama getters y setters).

Herencia

Nosotros sabemos que un auto es distinto de un colectivo, que a su vez es distinto de un tren, que a su vez es distinto de una moto, etc. Sin embargo, todos comparten algo: son medios de transportes terrestres. Acá es donde aparece la herencia. Voy a tener una **clase padre o superclase** a la que llamaré `TransporteTerrestre`, y otras cuatro **clases hijas o subclases** a las que llamaré `Auto`, `Colectivo`, `Tren` y `Moto`. Las clases hijas van a compartir todas algunos atributos y métodos, que heredarán de la clase `TransporteTerrestre` .

Es importante remarcar que a cada clase `Auto`, `Colectivo`, `Tren` y `Moto` la podemos visualizar con una imagen. Sin embargo, a la clase `TransporteTerrestre` no. En estos casos, decimos que la clase es abstracta. Las clases abstractas son aquellas que no se puede instanciar. En este ejemplo no tendría sentido crear un objeto de tipo `TransporteTerrestre` ya que no sería real.



Ahora bien ¿por qué utilizar la herencia? Para ahorrarnos de escribir mucho código. El objetivo de reutilizar lo que ya escribimos debe estar presente siempre. Ejemplo:

```
// TransporteTerrestre.h
class TransporteTerrestre {
    private:
        // Atributos
        string color, patente, marca;

    public:
```

```
        // Métodos
        void encender();
        void apagar();
        void acelerar();
        void frenar();
};
// Auto.h
class Auto: public TransporteTerrestre  {
    private:
        // Atributos
        int espacioBaul;
        bool techoDescapotable;

    public:
        // Métodos
        void cargarBaul();
};
```

Luego, la clase Auto tiene 5 atributos (`color, patente, marca, espacioBaul, techoDescapotable`) y 5 métodos (`encender(), apagar(), acelerar(), frenar(), cargarBaul()`)

Las formas de heredar son tres:

- `public` (la que en general se utiliza): lo público queda público, lo protegido protegido y lo privado privado
- `protected`: lo publico se convierte a protegido
- `private` (en general no se utiliza): se convierte todo a privado

Constructores y herencia

A diferencia de lo que ocurre con los métodos y atributos no privados, los constructores **no** se heredan. Cuando existe una relación de herencia entre dos o más clases y se crea un objeto de una clase hija, se ejecuta no sólo ese constructor sino también el de todas las padres de S. Para ello se ejecuta en primer lugar el constructor de la clase que ocupa el nivel más alto en la jerarquía de herencia y se continúa de forma ordenada con el resto de las subclases.

```
// Constructor de ClaseA
ClaseA(int x);

// Constructor de ClaseB
ClaseB(int x, int y);
```

Como `ClaseA` tiene un solo atributo, lleva un sólo parámetro. Por otro lado la `ClaseB` tiene dos atributos y recibe dos parámetros, pero uno de ellos es heredado de la `ClaseA`. Para que no haya problemas al compilar, el constructor de la clase padre debe llamarse antes que cualquier instrucción, por este motivo deben usarse inicializadores (instrucciones que van antes del cuerpo del método). Esto es así porque se crea en primer lugar, la porción del objeto de la superclase y luego la de las subclases. Al definir los constructores en el .cpp hay que escribir

```
// Constructor ClaseA
ClaseA :: ClaseA(int x) {
    this->x = x;
}

// Constructor ClaseB
ClaseB:: ClaseB(int x, int y) : ClaseA(x) {
    this->y = y;
}
```

Polimorfismo

El polimorfismo significa que un objeto puede tener varias (poli) formas (morfo). Es la propiedad que tienen distintos objetos de comportarse de distinta manera ante un mismo mensaje. Por ejemplo si a un objeto *Persona* le pedimos que se alimente, lo hará de manera distinta a un objeto *Perro*, y a su vez este lo hará de una manera distinta a un objeto *Pájaro*, y así sucesivamente. Por lo tanto el mismo método: *alimentar* trabajaría de forma diferente dependiendo qué objeto lo invoque. Esto puede ser obvio si se sabe cuál es ese objeto, pero la gracia del polimorfismo es no saber qué objeto lo está invocando.

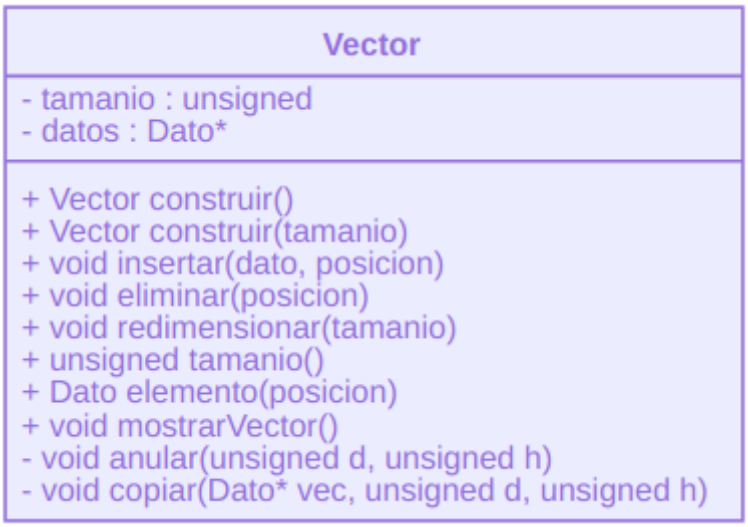
Nota: faltan cosas

Genericidad - TDA, Clase y Objeto

Introducción

La genericidad o condición de genérico la aplicaremos en todas las clases contenedoras, como vectores, listas, árboles, etc. Muchas veces veremos estas clases con tipos de datos enteros, como un `int` o `char` para concentrarnos en la estructura de datos y no en lo que guarda, pero la realidad es que en esas estructuras desearemos almacenar tipos diversos de datos. Para intentar entender que es la genericidad vamos a ver el siguiente ejemplo

TDA Vector



Clase Vector

Vector.h

```
#ifndef VECTORDINAMICO_VECTOR_H
#define VECTORDINAMICO_VECTOR_H

#include <iostream>
using namespace std;

// Acá puedo modificar cuál va a ser el tipo de Dato
typedef int TipoDeDato;
const int NULO = 0;

class Vector {

    // Atributos
    private:
        unsigned tamanio;
        TipoDeDato* datos;

    // Metodos
    public:
        // PRE: -
        // POST: construye un vector vacío
        Vector();

        // PRE: largo tiene que ser mayor que 1
        // POST: construye un vector de tamaño largo
        Vector(unsigned largo);

        // PRE:
        // POST:
        Vector(const Vector& vec);

        // PRE: -
        // POST: libera la memoria
        ~Vector();

        // PRE: dato debe ser un dato válido, 0 <= pos < tamanio
        // POST: inserta d en vector[pos]
        void insertar(TipoDeDato dato, unsigned pos);

        // PRE: pos debe ser un entero
        // POST: asigna NULO a vector[pos]
        void eliminar(unsigned pos);

        // PRE: -
        // POST: devuelve la longitud del vector
        unsigned tamanioVector();

        // PRE: pos debe ser un entero mayor o igual que 0
        // POST: devuelve el dato que esta en vector[pos]
        TipoDeDato obtenerElemento(unsigned pos);

        // PRE: largo debe ser un entero mayor que 1
        // POST: si largo > tamanio lo agranda conservando los datos anteriores
y agregando vacios
        //       si largo < tamanio lo achica y se pierden los datos que exceden
        //       si largo = tamanio no hace nada
        void redimensionar(unsigned largo);
```

```
        // PRE: -
        // POST: recorre el vector y lo muestra por pantalla
        void mostrarVector();

    private:
        // Coloca valores nulos al vector en las posiciones indicadas
        // PRE: 0 <= inicio <= final <= longitud del vector
        // POST: en datos coloca nulos desde d hasta h inclusive
        void asignarNuloAlVector(unsigned inicio, unsigned final);

        // Copia los datos del vector pasado por parametro desde d hasta h
        // PRE: 0 <= inicio <= final <= longitud del vector
        // POST: en datos coloca los valores del vector
        void copiarDatos(TipoDeDato* vec, unsigned inicio, unsigned final);
};

#endif //VECTORDINAMICO_VECTOR_H
```

Vector.cpp

```
#include "Vector.h"

Vector::Vector() : tamanio(0), datos(0) {}

Vector::Vector(unsigned largo) {
    tamanio = largo;
    datos = new TipoDeDato[largo];
    asignarNuloAlVector(0, largo);
}

Vector::Vector(const Vector& vec) {
    tamanio = vec.tamanio;
    if (tamanio > 0) {
        datos = new TipoDeDato[tamanio];
        copiarDatos(vec.datos, 1, tamanio);
    } else datos = 0;
}

Vector::~Vector() {
    if (tamanio > 0)
        delete [] datos;
}

void Vector::insertar(TipoDeDato dato, unsigned pos) {
    datos[pos] = dato;
}

void Vector::eliminar(unsigned pos) {
    datos[pos] = NULO;
}

unsigned Vector::tamanioVector() {
    return tamanio;
}

TipoDeDato Vector::obtenerElemento(unsigned pos) {
```

```
    return datos[pos];
}

void Vector::redimensionar(unsigned largo) {
    if (largo != tamanio) {
        TipoDeDato* auxiliar = datos;
        datos = new TipoDeDato[largo];
        copiarDatos(auxiliar, 0, largo);
        delete []auxiliar;
        if (largo > tamanio)
            asignarNuloAlVector(tamanio + 1, largo);
        tamanio = largo;
    }
}

void Vector::mostrarVector() {
    for (unsigned i = 1; i <= tamanio; i++)
        cout << datos[i] << " ";
}

void Vector::asignarNuloAlVector(unsigned inicio, unsigned final) {
    for (unsigned i = inicio; i <= final; i++)
        datos[i] = NULO;
}

void Vector::copiarDatos(TipoDeDato* vec, unsigned inicio, unsigned final) {
    for (unsigned i = inicio; i <= final; i++)
        datos[i] = vec[i];
}
```

Objetos

main.cpp

```
#include "Vector.h"

int main() {
    Vector vector(4);
    unsigned tam = vector.tamanioVector();

    for (unsigned i = 1; i <= tam; i++)
        vector.insertar(i+i, i);

    cout << endl << "Antes de la redimension: ";
    vector.mostrarVector();
    cout << endl;

    vector.redimensionar(7);
    cout << endl << "Agrando el vector: ";
    vector.mostrarVector();
    cout << endl;

    vector.redimensionar(5);
    cout << endl << "Achico el vector: ";
    vector.mostrarVector();
    cout << endl;
```

```
    vector.redimensionar(8);
    cout << endl << "Agrando el vector: ";
    vector.mostrarVector();

    return 0;
}
```

Cómo funciona el método redimensionar

Primero se fija que el largo al que se vaya a redimensionar sea distinto del tamaño actual del vector (de ser iguales no hace nada).

Luego se crea un puntero auxiliar, que apunta a la misma dirección que `datos`.

Ahora se reserva memoria en `datos` por lo que deja de apuntar al vector que teníamos, y apunta a esta nueva memoria.

Una vez hecho esto, se copian los datos que están en `auxiliar` (recordemos que son los *originales*) de manera que queden en la nueva memoria.

Lo que resta ahora que tenemos nuestros *datos originales* en la nueva memoria, es liberar la original, a la que se accede mediante el puntero `auxiliar`.

Finalmente si el vector se agrandó, rellena los nuevos espacios con NULO (si se achicó quedó recortado) y se le asigna al atributo `tamanio` el largo recibido por parámetro

Nota: la variable puntero auxiliar "muere" cuando se termina la función, el que "sobrevive" es el atributo de la clase, datos

Notas

1. Utilizamos un `typedef TipoDeDato` para que se pueda cambiar el tipo de dato del vector por un `char`, un `float`, etc. Por ese motivo se trabajó a lo largo de todo el código con `TipoDeDato` en lugar de `int`. Esta es la primera y tibia aproximación a la **genericidad**.
2. También debemos definir una constante NULO ya que el nulo puede variar no solo según el tipo de dato, sino según el tipo de problema con el que se trabaja.
3. Los métodos anular y copiar están privados porque no nos interesa en que el usuario los emplee. Pero son útiles para modularizar porque otros métodos los pueden utilizar, como el método de redimensionar y el constructor de copia.

Templates

En el código anterior se utiliza `typedef Dato` para poder modificar con facilidad el tipo de Dato del vector. Pero ¿qué sucede si necesitamos un vector de `char` y otro de `int`? ¿O si necesitáramos un vector de elementos más complejos, como los datos de un empleado (legajo, nombre, dirección, etc.)? No sería bueno en ningún método copiar o devolver un dato muy grande y complejo.

Un enfoque, teniendo en la mira el objetivo de la programación genérica, son las plantillas (templates). Los templates, en lugar de reutilizar el código objeto, como se estudiará más adelante en el polimorfismo, reutiliza el código fuente. ¿De qué manera? Con parámetros de tipo no especificado. Se debe agregar, antes de definir la clase la siguiente sentencia:

```
template < typename TipoDeDato >
class Vector {
    ...
};
```

Luego, cuando vayamos a utilizar el vector debemos indicarle de qué tipo es. Por ejemplo:

```
Vector< char > vecChar; // Vector de char
Vector< string > vecStr; // Vector de strings
Vector< string > vecStr2; // Otro vector de strings
```

En el ejemplo de recién, el compilador ve que necesita un vector para un tipo `char`, por lo que genera automáticamente el código por nosotros. Es decir reemplaza `TipoDeDato` por `char` en todas sus ocurrencias, generando el código completo. Luego, cuando compile la segunda línea, hace lo mismo pero con `string`.

Características

- Un template no existe hasta que se instancia. Es decir, hasta que no se crea algún objeto indicando el tipo no genera ningún código.
- Es indistinto utilizar la palabra `typename` o `class`. Pero se prefiere la primera ya que el parámetro no tiene por qué ser un objeto sino cualquier tipo.

Ventajas

- Generan código de forma automática por nosotros.
- Hay verificación de tipos. Esto, es una ventaja porque es una seguridad que el compilador verifique que los tipos sean correctos.
- Como la definición de tipos se resuelve en tiempo de compilación, los ejecutables son más rápidos que los generados utilizando herencia y polimorfismo.

Desventajas

- Se debe generar todo el código en el archivo punto h. Por lo tanto no se separarán las declaraciones por un lado (header) y las definiciones por otro (cpp). De todas formas, es recomendable seguir separando las declaraciones de las definiciones, aunque lo incluyamos todo en el mismo archivo.
- Se necesita indicar el parámetro cada vez que se indique la clase cuando se definen los métodos.
- Hay que sacar el método anular y la constante `NULO` dado que el valor nulo será distinto según el tipo de dato y el problema. Se verá como solucionar esto más adelante con el polimorfismo.