CONOCIMIENTOS PARA REALIZAR EL EXAMEN PRÁCTICO DE POO

Namespaces Declarar variables

~~Se utilizan para mantener el código limpio en caso de que el tamaño del proyecto crezca~~. Reemplazan a la idea de simplemente declararlos en el main o en la zona de funciones para mantenerlos más limpios. Se declaran y usan de la siguiente manera:

```
namespace LibraryA {
    double pi = 3.14159;
    double square(double number){
      return number * number;
}
}

namespace LibraryB {
    double pi = 3.14;
}

int main() {
    std::cout << "LibraryA's Pi: " << LibraryA::pi << std::endl;
    std::cout << "LibraryB's Pi: " << LibraryB::pi << std::endl;
    std::cout << "5 squared is: " << LibraryA::square(5) <<
std::endl;
    return 0;
}
```

String

Es parte de std → std::string text = “hola”; Y se puede concatenar haciendo: Std::string text2 = text + “mundo”. Necesita #include <string>

```
#include <iostream>
#include <string>
int main() {
    // Ejemplo de string.find() → busca el caracter o subcadena
dentro de dicha cadena
    std::string text = "hello world!";
    std::cout << "first letter of string: " << text[0] << std::endl;
    std::cout << "length of string: " << text.length() << std::endl;
```

```
    // Find the position of a substring
    size_t pos1 = text.find("world");
    // Find existe para otros tipos de variables!
    if (pos1 != std::string::npos) {
        std::cout << "'world' found at position: " << pos1 <<
std::endl;
    } else {
        std::cout << "'world' not found." << std::endl;
    }
    return 0;
}
```

string → int/float

También siguen existiendo stoi(string) o stof(string) para cambiar string a int o a float y viceversa usando to_string(int/float)

int/float → string

Class

Funcionan literalmente igual que un struct, ~~pero los struct solo guardan variables~~.
Permiten crear elementos con partes privadas (variables) y públicas (funciones)

función definida en 1-3 líneas se añade "inline"

```
Class point{
Private:
     Int x_, y_;
Public:
     Void Set(int x, int y) {x_=x, y_=y;}
inline Int Getx(){return x_;}
Int Gety(){return y_;}
}; //importante el ";" como si fuera un struct

Point p; //importante recordar que si quieres usar la clase tienes
que crear algo de esa clase
p.Set(69,-420);
cout<<p.Getx()<<"\n";
cout<<p.Gety()<<"\n";
//obviamente, cout <<p.x_; no funciona porque deniega el acceso
```

Si solo escribes el prototipo de la función, en el .cc deberás llamarlo así:

```
Void Person::Set(int x, int y){x_ =x, y_=y;}
```

Constructor: te permite que, cuando llames a una clase, definir ~~le puedas meter~~ directamente los parámetros de sus variables. Se hace con una función con el mismo nombre de la clase. ~~Se le pueden poner datos por defecto igualando las variables a algo. OJO! Poner datos por defecto indica que cuando crees una instancia de la clase, no es obligatorio poner ese dato, ya que si no lo pones, se vuelve por defecto. Aquellos datos que no tengan por defecto SON OBLIGATORIOS (esto suele ser el id o el dni)~~

```
class Person {
private:
    std::string name;
    int age;

public:
    // Parameterized constructor with default settings. Remember that
every variable with a default setting must go AFTER those who don't
have a default setting.
    Person(const std::string& personName = "empty", int personAge =
0) {
        name = personName;
        age = personAge;
    }
};

int main() {
    // Create the object and initialize it in a single step
    Person p1("Alice", 25); // if this were empty, it would look like
this: person.personName() = empty, person.personAge = 0
    return 0;
}
```

Recuerda que, si en vez de escribir todo el constructor en el .h, decides solo poner su prototipo, en el funciones.cc debes llamar a esa función como Person::Person.

```
Person::Person(std::string id, std::string name, std::string town,
           std::string province, std::string country,
           int age, double rank, int entry_year){
id_=id;name_=name;town_=town;province_=province;country_=country;
           age_=age;rank_=rank; if(entry_year<2000){entry_year_ = 0;}
else{entry_year_=entry_year;}}
```

NO HACER :)

Si te da por iniciar algún miembro puedes hacerlo

```
class A{
Private:
      Int x_, y_;
Public:
      A(): x_(1), y_(4);
}
```

Guardas de inclusión

~~Hermano, ponlas y ya. Solo en los .h~~

```
#ifndef PERSON_H
#define PERSON_H

//bunch of code

#endif
```

Bool t/f

~~Trae true o false como respuesta, yknow~~

```
bool found;
if(found){ // if true
Cout << "kill" << endl;
}
if(!found){ // if false
Cout << "self" << endl;
}
```

Const

Pa declarar constantes, ~~solo se pone "const" antes de la cosa. Por convención se pone k al principio del nombre de la variable para explicar que es constante~~

```
const int kPi = 3.141592;
```

Vector

~~Se necesita #include <vector> para que funcione, y se declara como std::vector<int> V~~

Puedes editar lo que hay en un vector usando V[5] = 0 o V.at(5) = 0 pero .at() maneja errores así que ~~simplemente úsalo porque~~ es mejor ~~y ya~~. Se almacena en el heap ~~btw~~

```
#include <vector>
Int main(){
      std::vector<int> V = {0, 1, 2, 3, 4}
      V.at(4) = 5;
//now the vector will look like this: 0 1 2 3 5
}
```

El tamaño del vector es dinámico. Tanto en vectores como en arrays el tiempo de acceso es O(1). Tienes la función V.push_back(9) que mete un 9 en la última posición del vector, V.pop_back() que elimina la última cosa del vector y V.clear() para los ragequits.
Puedes por alguna razón crear un vector con todos los elementos iguales.

```
std::vector<int> v(n,10) //vector with n size and all values as 10
```

~~Recuerdas el coñazo de organizar un vector? Worry not my child~~:

```
sort(v.begin(), v.end())
```

Referencias

~~Los punteros de C, pero con menos quebradero de cabeza~~. Hace que lo que pases a las funciones sea el valor literal, en vez de crear una copia. ~~Esto es cuando quieres alterar el valor de una variable cuando pasa por una función~~. Se ahorra memoria, tiempo de cómputo y se simplifica el código

```
#include <iostream>
using namespace std;
//example using references. & is needed when calling the function
void intercambia(int &a, int &b) {
    int aux = a;
    a = b;
    b = aux;
}

int main() {
    int x = 5, y = 10;

    cout << "Before swapping: x = " << x << ", y = " << y << endl;

    intercambia(x, y);  // Call the function with references
```

```
    cout << "After swapping: x = " << x << ", y = " << y << endl;

    return 0;
}
```

También existe const &a, se usa cuando no se quiere modificar la variable, pero sí leer o acceder para cualquier cosa. ~~Muy útil si sin querer te cargas todo el código pasando mierdas por referencia :3~~

Array

Estático y se almacena en pila. Muy útil para matrices. Necesita #include <array>

```
std::array<int, 3> a = {1,2,3} //array de 3 enteros
std::array<std::array<int,3>,3> ar = {{{1,2,3},{4,5,6},{7,8,9}}}
//matriz de enteros 3x3
```

Iterator

Son objetos que se usan para recorrer elementos de lo que esta peña llama "colecciones" pero son vectores, arrays y listas. Cada colección tiene un iterador begin() y end() ← más allá del último elemento, no es el último elemento. Necesita #include <iterator> opcionalmente.

```
//ejemplo de función que usa un iterador
void Market::mostrarClientes(){
    std::list<Client>::iterator it_lista;
//haces std::(sobre qué va a iterar)::iterator
        for(it_lista = client_list_.begin(); it_lista !=
client_list_.end(); ++it_lista){
            std::cout << (*it_lista).GetId() << "\t" <<
(*it_lista).GetName() << std::endl;
        }
}
```

Date cuenta de que para el cout, el iterador es un puntero, se debe derreferenciar para hablar del contenido donde está puesto

Auto

Se encarga de traducir automáticamente el tipo de variable con lo que lo inicializar.

```
auto i = 10; //int
auto pi = 3.14; //float
auto perchance = true; //bool

std::vector<int> v = {1,2,3,4,5};
auto it = v.begin(); // std::vector<int>::iterator
//ESTO ES MUY ÚTIL POR QUÉ NO NOS LO ENSEÑARON DIRECTAMENTE
```

El código de antes se puede traducir a esto:

```
void Market::mostrarClientes(){
        for(auto it_lista = client_list_.begin(); it_lista !=
client_list_.end(); ++it_lista){
            std::cout << (*it_lista).GetId() << "\t" <<
(*it_lista).GetName() << std::endl;
        }
}
//no declaramos el iterador, sino que lo ponemos como auto en el for
```

NO HACER :)
Forma épica de ciclar un loop (solo que sin iteradores, si los piden esto no vale)

```
std::vector<int> numeros = {1, 2, 3, 4, 5};
for (const auto& numero : numeros) {
    std::cout << numero << endl;
}
//const auto& es referencia constante, no crea copias y no modifica
los elementos. Obviamente si lo que necesitas es modificar los
elementos no pones const
//"numero" debe ser del tipo que sean los elementos de "numeros", ya
que cada elemento se está metiendo en "numero" una única vez
```

List

Se inicializa de la misma forma que un vector: list<type> name; Necesita #include <list>
Tiene las mismas cosas que un vector la verdad:

```
list<int> listy;
listy.push_back(1);
listy.push_front(0);
for(auto it = listy.begin(); it!=listy.end(), it++){
   cout << "killSelf" << endl;
```

```
}
```

Map

Necesita #include <map>

Funciona de la siguiente manera: es una relación entre una clave y su correspondencia. Debes indicar qué tipo es la clave y la correspondencia. Por ejemplo, si quieres un mapa que relacione ids con cantidades del producto con ese id, haces:

```
std::map<std::string,int> product_quantity_;
```

si es 1 existe, 0 no

Y se pueden hacer cosas como .count(), aunque como solo hay una clave de cada tipo, solo devuelve o cero o uno. Aquí añadimos un producto comprobando si primero existe la clave del producto en el mapa. En caso negativo la añadimos. En caso afirmativo, simplemente sumamos 1 a la correspondencia.

```
void Basket::AddProduct(Product p1){
      if(product_quantity_.count(p1.GetId()) ==0){
            product_quantity_[p1.GetId()] = 1;
            product_list_.push_back(p1.GetId());
      }
      else{
            product_quantity_[p1.GetId()]++;
      }
      total_+=p1.GetPrice();
}
```

si existe +1

si no existe =1 (0+1)

Para meter algo en el mapa, haces nombreMapa[clave] = correspondencia. Como para cada pareja hay dos elementos, puedes acceder a cada uno haciendo nombreMapa.first para la clave o nombreMapa.second para la correspondencia

Ejemplo completo:

```
#include <iostream>
#include <map>
using namespace std;

int main() {
    // Create a map
```

```
    map<string, int> age;

    // Insert elements
    age["Alice"] = 25;
    age["Bob"] = 30;
    age["Charlie"] = 22;

    // Access elements
    cout << "Alice's age: " << age["Alice"] << endl;

    // Iterating through the map
    for (const auto &pair : age) {
        cout << pair.first << " is " << pair.second << " years old."
<< endl;
    }

    // Find an element
    auto it = age.find("Bob");
    if (it != age.end()) {
        cout << "Found Bob, age: " << it->second << endl;
//como el iterador es puntero, se usa -> porque no es literalmente el
elemento en sí
    } else {
        cout << "Bob not found!" << endl;
    }

    return 0;
}
```

A los maps también se les puede hacer iteraciones, recordando que tienen fisrt y second:

```
for(auto it=mymap.begin(); it!=mymap.end(); ++it){
cout << it->first << " => " << it->second << '\n';
}
```

Enum class

Sirve para enumeraciones, donde cada cosa de la enumeración tiene un valor asignado

```cpp
enum class Color {
    Red,
    Green,
    Blue
};

int main() {
//se pueden crear variables de ese tipo
    Color favorite = Color::Green;
    // Scoped access
    if (favorite == Color::Green) {
        cout << "Favorite color is Green!" << endl;
    }
    // Cannot implicitly convert to int
    // cout << favorite;  // Error
    return 0;
}
```

También se pueden asignar valores dentro de la clase y usarse de esta forma:

```cpp
//asignamos a las direcciones valores
//aquí especificamos que los valores que asignamos son int (Direction
: int)
enum class Direction : int {
    North = 1,
    South = 2,
    East = 3,
    West = 4
};
//creamos un switch que dependa de las direcciones
void navigate(Direction dir) {
    switch (dir) {
        case Direction::North:
            cout << "Going North!" << endl;
            break;
        case Direction::South:
            cout << "Going South!" << endl;
            break;
        case Direction::East:
```

```
            cout << "Going East!" << endl;
            break;
        case Direction::West:
            cout << "Going West!" << endl;
            break;
    }
}
int main() {
    Direction myDirection = Direction::East;
    navigate(myDirection);
    return 0;
}
```

Sobrecarga de funciones

Básicamente es crear funciones con el mismo nombre si lo que van a hacer es parecido pero solo cambia lo que se va a mandar como parámetro.

```
// Function overloading: same name, different parameter lists
void print(int num) {
    cout << "Integer: " << num << endl;
}

void print(double num) {
    cout << "Double: " << num << endl;
}

void print(string str) {
    cout << "String: " << str << endl;
}

int main() {
    print(42);           // Calls print(int)
    print(3.14);         // Calls print(double)
    print("Hello!");     // Calls print(string)

    return 0;
}
```

Herencia

Básicamente es crear una clase ~~desde otra trayendo~~ que tenga todas las cosas que tenía ~~esa~~ otra

```
//en vehicle.h
class Vehicle{
public:
  void SetWheels(int w);
  int GetWheels();
  void SetPassenger(int p);
  int GetPassenger();
private:
  int wheels_;
  int passengers_;
};
```

```
//en truck.h
#include "vehicle.h"
/*al declarar una función heredada de otra, haces:
class name : public name2*/
class Truck : public Vehicle{
public:
  void SetLoad(float l);
  float GetLoad();
private:
  float load_;
};
```

```
//desde el main
#include "truck.h" //esto incluye truck, que incluye vehicle.h
int main()
{
Truck c;
cout << "ruedas del camión = " << c.GetWheels(); //función heredada
cout << "carga del camión = " << c.GetLoad(); //función propia
return 0;
}
```

Pero lo normal no es el ejemplo anterior. Lo normal es que tu primera clase tenga un constructor con algún parámetro obligatorio y alguno opcional:

```
class Person{
      private:
std::string id_;
            std::string town_;
            std::string province_;
            std::string country_;
            std::string name_;
            int entry_year_;
            int age_;
            double rank_;
public: //constructor
      Person(std::string id, std::string name = "empty", std::string
town= "empty",std::string province= "empty", std::string country=
"empty", int age= 0, double rank = 0, int entry_year = 0);
//este constructor tiene todos los parámetros opcionales salvo el id
}
```

Y que tú, al crear una clase heredada, debas mandarle esos parámetros:

```
class Client: public Person{
    private:
        int premium_; //dato privado de client
    public:
        Client(std::string id, std::string name="empty", std::string
town="empty", std::string province="empty", std::string
country="empty", int age=0, double rank=0.0, int entry_year=0, int
premium=0): Person(id, name, town, province, country, age, rank,
entry_year){premium_=premium;}
/*este constructor recoge todos los datos (si alguno es
obligatorio/opcional lo pone igual que está en person) pero al final
pone ": Person(...)", es una forma de decir "de todo lo que he
recogido, a person va lo siguiente"*/
        int GetPremium(){return premium_;} //función de client
        void SetPremium(int premium){premium_=premium;} //función de
client
};
```

This (puntero)

El puntero “this” se manda de manera implícita a todos los métodos (funciones) de una

clase.

```
#include <iostream>
using namespace std;

class Vehiculo {
private:
    int ruedas_;
public:
    void setRuedas(int newRuedas) {
        this->ruedas_ = newRuedas;  // Asigna el valor usando this.
//es lo mismo que hacer ruedas_ = newRuedas sin más, el this lo
usamos por si hay ambigüedad.
    }

    int getRuedas() const {
        return ruedas_;  // Devuelve el número de ruedas.
    }
};

int main() {
    Vehiculo coche;
    coche.setRuedas(4);  // Configura el número de ruedas a 4.
    cout << "El coche tiene " << coche.getRuedas() << " ruedas." <<
endl;

    return 0;
}
```

Esto va a ser de más utilidad luego.

Sobrecarga de operadores

Permite redefinir lo que hacen los operadores clásicos +, -, *, ==, etc al usarlos una clase específica. Se tiene que nombrar una función de operador (es decir, poner la palabra operator) y sin espacios, poner qué operador es el que vamos a redefinir. Es decir, operator+, operator++, operator=...

```
//Ejemplo en el que hacemos sobrecarga de operadores y de funciones.
Tenemos dos funciones operator=, una de ellas iguala el valor inicial
a un entero, y otra copia todos los datos de otro operador
//importante recordar que estamos poniendo Contador Contador porque
```

```
esto es el .cc, en el .h ponemos solo "Contador operator=(int i);"
Contador Contador::operator=(int i){
      if(i < minValue_){
            initValue_ = minValue_;
      }
      else if(i > maxValue_){
            initValue_ = maxValue_;
      }
      else{
      initValue_ = i;
      }
      return *this;
}
//Devolvemos *this en ambos casos porque queremos mandar todo el
objeto con el cual estamos operando (this es un puntero, por eso
derreferenciamos). La verdad, solo he visto *this usado en sobrecarga
de operadores así que tampoco es para rallarse mucho, es lo que
devuelves y ya.
Contador Contador::operator=(Contador cont){
      minValue_ = cont.minValue_;
      initValue_ = cont.initValue_;
      maxValue_ = cont.maxValue_;
      return *this;
}
```

NO HACER :)

Dicen los apuntes que por conveniencia hacer un constructor copia por defecto. Eso sería escribir en el .h "Contador(const Contador&) = default" ← cambiar "contador" por el nombre de la clase, yknow. (pero vamos que a mí los ejercicios me han funcionado sin esto)

En algunas funciones como ++ y --, se pueden hacer dos cosas, ++i ó i++.

```
//Ninguna función recibe nada, pero para que no sean iguales, a la de
abajo se le pone int, pero es falso, lol.
//++i. Modifica directamente el valor del objeto
Contador Contador::operator++(void){
      if(initValue_+1 <=maxValue_){
            initValue_++;
      }
```

```
        return *this;
}
//i++. No se puede modificar directamente el valor del objeto, así
que pasamos el objeto al que apunta “this” a aux de tipo contador, y
es eso lo que devolvemos.
Contador Contador::operator++(int){
        Contador aux = *this;
        if(initValue_+1 <=maxValue_){
                initValue_++;
        }
        return aux;
}
```

Hay algunos operadores (vamos, << y >>) que necesitan la siguiente forma:

```
//se necesita ostream para sacar datos por pantalla, istream para
introducirlos por teclado
//En contador.h
friend std::ostream& operator<<(std::ostream &stream, Contador
&cont);
friend std::istream& operator>>(std::istream &stream, Contador
&cont);
//en contador.cc
std::ostream& operator<<(std::ostream &stream, Contador &cont){
        stream << cont.initValue_ << "\n";
        return stream;
}
std::istream& operator>>(std::istream &stream, Contador &cont){
        int value;
        bool valid=false;
        while(valid == false){
                std::cout << "Introduzca un valor para el contador: ";
                stream >> value;
                if(value < cont.minValue_ || value > cont.maxValue_){
                        std::cout << "Ha ocurrido un error. Valor
inválido\n";
                }
                else{
                        cont.initValue_=value;
```

```
                valid=true;
            }
        }
        return stream;
}
```

Funciones Friend

Son funciones que se declaran dentro de la clase pero no pertenecen a esta. Estas funciones pueden acceder directamente a las variables en private. Más fácil con un ejemplo: (voy avisando, estas funciones se usan con sobrecarga de operadores)

```
//contador.h
class Contador{
private:
        int initValue_;
        int minValue_;
        int maxValue_;
public:
        Contador(int initValue_ = 0, int minValue_ = 0, int maxValue_ =
1000);
        int get(){return initValue_;}
        friend Contador operator+(Contador cont, int i);
        friend Contador operator+(int i, Contador cont);
};

//contador.cc
//Aquí no ponemos Contador Contador, porque no es un método de
contador, incluso aunque se escriba dentro de la clase. Por eso
mismo, tampoco podemos devolver "*this", porque si no perteneces a la
clase no lo tienes. Lo que hacemos es simplemente devolver un
contador con los datos metidos en el constructor.

Contador operator+(Contador cont, int i){
//aquí podemos tocar los datos privados directamente
        if(cont.initValue_ + i <= cont.maxValue_){
                return Contador(cont.initValue_ + i, cont.minValue_,
cont.maxValue_);
        }
```

```
      else{
            return Contador(cont.maxValue_, cont.minValue_,
cont.maxValue_);
      }
}
//esto es lo mismo pero al revés pa que la suma funcione en ambas
direcciones yknow
Contador operator+(int i, Contador cont){

      if(cont.initValue_ + i <= cont.maxValue_){
            return Contador(cont.initValue_ + i, cont.minValue_,
cont.maxValue_);
      }
      else{
            return Contador(cont.maxValue_, cont.minValue_,
cont.maxValue_);
      }
}
```

En main, como estas funciones devuelven un contador, para saber el resultado tendrás que hacer algo como:

```
Contador c(3,0,10) //contador entre 0 y 10 con valor inicial 3
Contador ResultadoSuma = c + 4;
Cout << ResultadoSuma.get() <<endl; //salida: 7
```

Plantillas

~~Imagina que eres un vago y quieres que~~ tu código imprima todos los datos de un vector independientemente de si el vector es de int, float, char* o lo que sea. ~~Estás de suerte amigo porque pa eso están las plantillas yatusabe queloque~~

```
#include <iostream>
#include <vector>
//se define de la siguiente forma:
template <class T>
//aquí decimos que el vector tiene elementos de tipo T (cualquiera)
void imprimir(const std::vector<T>& v) {
    for (size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " , ";
    }
```

```
    std::cout << std::endl;
}

int main() {
    std::vector<int> a = {1, 3, 5, 7, 9};
    std::vector<float> b = {5.6, 7.8, 3.9, 1.2};
    std::vector<char> c = {'h', 'o', 'l', 'a'};

    std::cout << "vector de enteros: ";
    imprimir(a);

    std::cout << "vector de floats: ";
    imprimir(b);

    std::cout << "vector de char: ";
    imprimir(c);

    return 0;
}
```

También puedes decidir que quieres que una clase que has creado haga lo mismo en sus métodos, es decir, que puedas crear una clase cuyos datos puedan ser int, floats y operar con ellos

```
#include <iostream>

template <class T>
class MiClase {
private:
    T x_, y_;
public:
    MiClase(T a, T b) : x_(a), y_(b) {}

    T Div() {
        if (y_ != 0) {
            return x_ / y_;
        } else {
            std::cout << "No se puede dividir por 0.";
        }
```

```
    }
};

int main() {
//aqui le decimos a la clase de lo que va a ser la clase, yknow.
    MiClase<int> iobj(10, 3);
    MiClase<double> dobj(3.3, 5.5);
//y al mandar la info, se interpreta automáticamente en T
    std::cout << "división entera = " << iobj.Div() << std::endl;
    std::cout << "división real = " << dobj.Div() << std::endl;
    return 0;
}
```

Ficheros (lo único que pide de ellos)

~~De ficheros pone "ver apuntes" y luego no hay apuntes de ficheros~~.
~~En fin, solo hay que saber lo siguiente:~~
#include <fstream> ~~pa que funcione~~
std::ofstream f("nombre.extensión") pa abrir un fichero con nombre y extensión (ventas.txt, por ejemplo)
if(f.is_open()){} ← ~~aquí escribes el código~~ ~~en caso de que el fichero haya abierto correctamente. En un else pones algo de que no se ha abierto bien o yo que sé~~.
f << texto; ← así metes texto en el fichero
f.close() ← lo pones al final para cerrar el fichero. ~~Recuerda que si no haces esto te ejecutan públicamente y hazme caso que yo estaré allí para aplaudir~~.

~~En fin, si te ha servido échate un follow, *que es gratis*~~