I/O en C

Printf y scanf\_s->#include<stdio.h>

C++

Cout,cin ->#include<iostream>

std::cout << "Write value 6" << std::endl;

con una variable a=3

std::cout << “A is “ << a << “\n”; (std:endl==”\n”)

std::cin >> a;

TECLAS:

F5: compilar y ejecutar. Debug(preview, testear pero ocupa más) o reléase(aplicación optimizada, usar al entregar)

Ctrl+k->ctrl+c poner comentarios

Ctrl+k->ctrol+u quitar comentarios

DEBUG:

Click izq a la izquierda de los números en la línea que queremos poner/quitar un beakpoint. Cuando se para le damos f5 para que tire hasta el siguiente breakpoint o f10 para que vaya de uno en uno.

Podemos ver el valor de las variables si pongo el ratón encima de la variable.

Si ponemos un breakpoint en el main en una función nos la pasará entera. SI queremos entrar dentro y ver qué pasa dentro ponemos el breakpoint en la función. Con F11 podemos entrar a la función desde la llamada a la función del main.

**using namespace std;** (después de los includes y antes que funciones) para que no haga falta poner los std:: todo el rato.

Ej. Cout<<”introdudsudiudis” <<endl;

Cin>>a;

---13-02

Struct vec2D{ main->

Float vecx; struct vec2D v; (ahora vec2D se escribe

Float vecy; como v)

}; v.vecx=1.0;

v.vecy=2.0; // v=(1.0 , 2.0)

---17-02

Char name[20]=”Ricard”; = char\* name=”Ricardo”;(de esta forma no hace falta saber cuanto va a ocupar el nombre, lo calcula solo cuando lo ponemos)

---24-02

**Gestión de memoria**

C nos ofrece 2 operadores para trabajar con memoria.

& => reference operator => devuelve una dirección de memoria

\*=> dereference operator => devuelve en valor guardado en una dirección de memoria.

int val = 5;

cout << &val<<endl; // nos da el número de memoria 0074F7AC ...

cout << \*(&val)<<endl; // nos da 5

**Punteros**

Puntero: tipo de una variable a la que le asignamos una dirección de memoria.

Sintaxis: \_\_\_\_\* \_\_\_\_\_ = \_\_\_\_;

Tipo del valor al que apunta\* nombre variable=dirección de memoria;

(tipo de puntero)

int val = 5;

int\* ptrVal = &val; \*Tipo puntero guarda un valor de dirección de memoria

int val = 5;

int\* ptrVal = &val;

cout << &val<<endl; // dirección val

cout << \*(&val)<<endl; // 5

cout << \*ptrVal << endl; // 5

cout << ptrVal << endl; // dirección val

operador “dereference”:

-Consultar el valor guardado en una @mem (@mem=dirección de memoria)

-Asignar (“cambiar”) el valor guardado en una @mem.

\*(@mem)= nuevo valor; //\*(&\_\_\_) o puntero que apunte a variable

**Inicialización de punteros**

1-Asignar una @mem. Int val=5;

Int\* ptrval=&val;

2-Inicializar a nulo. Int\* ptrval=0; // = nullptr; = NULL; (se puede usar cualquiera de las 3)

O si la variable ya está declarada: ptrval=0/nullptr/NULL;

Int vec[3]={1,2,3}; //punteros con vectores

Int\* ptr=&vec[1];

Deberes: Función que tiene que crear array (5 elementos int) inicializándola con los valores que queramos. 1.Imprimir posiciones de memoria de cada elemento 2.Utilizando \* sumar 1 a cada posición del vector(1,2,3,4,5🡪2,3,4,5,6) y print de todo el vector

---27-02

Sizeof(\_\_\_) me da cuantos bytes guarda en memoria:

int val = 5;

unsigned short a = 1;

cout << sizeof(val) << sizeof(a) << endl; // nos da->42 4bytes por int

y 2 por unsigned short

**Memoria**

Conjunto de celdas de 1byte que contienen una dirección cada una. Un int ocupa 4 bytes, pues ocupa 4 celdas con el número. Un puntero también ocupa 4 pero en vez de guardar un valor, guarda la dirección de memoria a **la primera celda**.

Arrays

Int vec[3]={1,2,3};

El nombre del array devuelve la @mem del 1r elemento (vec[0] en este caso)

cout << &vec[0] << "="<<vec<<endl;

Entonces si queremos el primer valor del array también lo podemos pedir haciendo \*vec ya que apunta a la primera casilla del vector.

int\* vecptr = vec; == int\* vecptr2 = &vec[0]; vec=&vec[0]

Los elementos del array se guardan en celdas consecutivas de la memoria.

Operaciones con punteros

[ ] -> funciona como un array

Int vec[]={1,2,3};

Int\* ptr=vec;

Ptr[0]

Ptr[1]

+ -> sumar/restar posiciones que queramos, incluido ++ y – para las consecutivas.

-. ->

int vec[] = { 1,2,3 }; Int\* &vec[0]; == vec

int\* ptr = vec;

for (int i = 0; i < 3; i++) {

cout << ptr[i] << endl; == cout << \*(ptr+i) << endl;

}

**Char\***

Podemos usarlo para strings.

Char\* Word = ”Hello”; no hace falta determinar el tamaño.

También podemos comparar con el ==

Se comporta como un array, por lo tanto:

- Podemos utilizar el [ ].

- Añade el ‘\0’ al final del array (“string”).

- Nombre del string devuelve @mem del 1r elemento.

-Podemos utilizar todos los operadores de punteros.

Deberes viernes: acabar ej y hacer del pdf el 2 y 3.

---03-03

Pase de parámetros con punteros

Hasta ahora, los parámetros de las funciones/acciones eran “copias”🡪(Paso de parámetros por copia)

Si modificamos el valor de la variable en la función, solo cambia dentro de la función (el valor de la copia)

Parámetros punteros: dentro de la función podemos modificar el valor de la variable pasando su @mem (pasando el parámetro como puntero)

¿Cómo le pasamos un array a una función?

Pasamos la @mem (“puntero”) al primer elemento del array.

int\* vecptr = &vec[0]; // o =vec;

printVec(vecptr); // o (vec) y borrando la linea anterior

//cout << vec[i] << endl; = cout << \*(vec + i) << endl; sin usar el operador [].

Punteros con tipos de datos propios

Struct Pair{ int main(){

Int x; struct Pair p;

Int y; p.x=0;

} p.y=0;

Struct Pair\* ptr=&p;

Ptr->x=2; no podemos usar el . con punteros,

hacemos una flecha ->

Memoria dinámica

Stack-Heap

Stack: lo lleva la maquina, mas eficiente y rápido. Donde se guardan ints, floats … que declaramos normalmente.

Heap: Gestionamos nosotros, no tan eficiente pero mas flexible.

Nosotros tocamos el heap. Luego de usarlo tenemos que liberarlo para eviar memory leaks.

Instrucciones para gestionar heap

Concepto previo: void\* (punteros ‘sin tipo’ predefinido)

Sirven para guardar la @mem de cualquier tipo de dato:

Float val =3,3;

Void \*ptr=&val;

Int val2=5;

Void\* ptr2=&val2;

Cout<<\*ptr2<<endl; esta no te la dejara usar si está declarado como void pointer

Cout<<\*((int\*)ptr2)<<endl; tenemos que decirle antes que es un int

Instrucciones:

**Reservar memoria:**

**Malloc**🡪”allocatar” un bloque de memoria de un tamaño que especificamos nosotros.

**Calloc**🡪”allocatar” un bloque de memoria de un tamaño que especificamos pero inicializa ese conjunto de celdas con 0’s.

Estos 2 devuelven void\*

**Liberar memoria:**

**Free**🡪”desallocata” un bloque de memoria

Ejemplo:

Int main(){

//Int vec[3]={1,2,3}; // esto va al stack pq no le decimos a la maquina nada especifico

int\* vec= (int\*)**malloc**(3\*sizeof(int)); //reservamos en memoria espacio para 3 enteros

vec[0]=1;

vec[1]=2;

…

Int\*vec2=(int\*)**calloc**(3,sizeof(int));

Free(vec);

Free(vec2);

System(“pause”);

Return 0;

}

Viernes test de punteros

Lunes clase repaso ej que no sepamos, dudas…

---06-03

Con cin no podemos usar char\*

Si usamos en un for - - i envez de i - - quiere decir que decrementa antes de la vuelta

cout << \*(array+i) << endl; ==== cout << array[i] << endl;

---13-3

Memoria dinámica c++ (‘alocatar’ y ‘liberar’ memoria en el HEAP

Tenemos 4 opciones

1. Int\* var=(int\*)malloc(sizeof(int);

.

.

free

Free(var);

1. Int\* var=(int\*) calloc(1, sizeof(int));

.

.

Free(var);

1. Int\* var=new int; (no inicializa, como malloc)

.

.

Delete var;

delete

1. Int\* vec=new int[4];

.

.

Delete[] vec;

Enums y structs no hace falta poner “struct NombStruct n;”, se puede poner “NombStruct n;” y ya.

Keyword “const”

Sirve para decir que el valor de: una variable (o puntero), retorna una función, un parámetro o un objeto de una clase es constante (su valor no se puede modificar).

Una variable, puntero u objeto etiquetados como const, se han de inicializar cuando se declaran y no se puede cambiar su valor-> si lo intentamos ERROR DE COMPILACIÓN.

Cuando declaras una variable de una clase se llama objeto o instancia (¿)

Const type\* name : pointer a una constante.

Type\* const name : puntero constante .

Const type\* const : puntero constante a una variable constante.

Referencias

Int val=3;

Int& alias=val; si yo ahora hago val=5 o alias=5, los dos cambiarán; son exactamente lo mismo.

(main)

doubleVal(a);

(función)

void doubleVal(int& param)//envez de hacer una copia de a y que a se quede igual,

{ cuando cambiamos param, la a también cambia

param \*= 2;

}

Clases

class Point {

private:

int x, y;

public:

int getx() { return x; }

int gety() { return y; }

};

Dentro de las clases tenemos atributos y métodos(funciones).

Para acceder desde fuera a una clase usamos los métodos. Yo puedo decidir que “visibilidad” tienen los elementos. Los private no los pueden cambiar des del main (sean atributos o métodos). Los public son accesibles des del main.

-----3 ej de cons y referencia

---17-03

Clases C++

Constructor de la clase: método “especial” que tienen todas las clases y que sirve para inicializar los atributos de la clase.

Tiene el mismo nombre que la clase y no devuelve nada, simplemente se ejecuta cuando declaramos objetos (variables) de esa clase.

Tenemos 3 tipos:

-Constructor por defectos: nos lo proporciona por defecto, y si no nos lo piden, no hace falta implementarlo.

Soldier();

-Constructor por defectos: le pasamos como parámetros los alores con los que queremos inicializar los atributos de la clase

Soldier(char\* name, int lifUnits);

-Constructor por copia: le pasamos un objeto de la misma clase, e inicializa los atributos con valor de los atributos de

Soldier(Soldier soldado);

Destructor: se encarga de “destruir” el objeto-> liberar memoria usada por el objeto

-Si no se “alocata “ memoria dinámica no hace falta hacer nada.

-Si en el constructor hemos alocatado memoria dinámicamente, aquí hemos de liberarla.

-Sintaxis: ~nombre\_de\_la\_clase();

-C++ también nos lo ofrece por defecto, por lo que si no hemos de liberar memoria dinámicamente, no hace falta ponerlo.

Soldier(char\*name, int lifeUnits) :name(name), lifeUnits(lifeUnits){}

Esto lo utilizaremos siempre que tengamos un atributo const ya que no podremos usarlo para asignar como hacemos normalmente.

This-> es un puntero al propio objeto. Lo usamos entre otras cosas para desambiguar.

Sobrecarga de operadores: en una clase puedo definir los operadores básicos: ==, ¡=, <,>,>=,+…

Tipo de retorno operator == (objeto de la clase);

Como usar “const” en los métodos de clase:

Int func(**const** Soldier s)

**Const** int func(…) se usan para cosas diferentes, el valor entero q me retorna esta función no se podrá modificar, no tiene q ver con q no pueda modificar el soldier.

Const int func(const Soldier s) **const** El const detrás de un método de una clase indica que en ese método no se puede modificar el valor de los atributos de la clase

--clases ej2

---27-03

**Métodos inline**

Para hacer un speed up de funciones/métodos pequeños (“con pocas líneas de código”)

¿Cómo? => usando la keyword **inline**

El compilador “reemplaza” la declaración de las funciones inline por su código en tiempo de compilación, en vez de “ir al código” en tiempo de ejecución.

PROS:

-speed up del programa.

-cuando se llama la función, se mejora la gestión en la stack de variables.

CONS:

-mayor tamaño del ejecutable.

-si cambiamos el código se ha de recompilar.

-exes(ejecutables) grandes pueden traer problemas con la gestión de memoria.

Cuando implementamos métodos en la declaración implícitamente os trata como “inline”.

\*Usamos inline con métodos pequeños\*

Donde ponerlo:

Inline tipo de retorno nombre método (…);

Lo que hay dentro de la declaración es inline por defecto

**Templates**

Nos permiten definir funciones/acciones con un tipo genérico

Independientemente del tipo

Sintaxis:

Template<class type>

Podemos poner el nombre que queramos (como si ponemos pepe)

Type sum(type a, type b){

Return a+b;

}

Llama desde el

Int main(){

.

.

Int res=sum<int>(3,4),

.

.

}

--homework templates 1

---31-03

**Clases con templates**

Del mismo modo que hacemos funciones para “tipos genéricos” podemos usar templates para usar “tipos genéricos” con clases.

**Cuando usamos templates con clases, tenemos que implementar todo en el .h.**

Sintaxis

Template<class Type>

Class Point2D{

Private:

Type coords[2];

Public:

Point2D(Type x, Type y){

Coords[0]=x;

Coords[1]=y;

}

Declaración de objetos

Point2D<int> p(115,36);

//mirar el codigo q nos pasa el para saber como implementar fuera de la clase

**Herencia**

La herencia en C++ requiere una clase base(padre) y derivada(hijo). La/s clase/s derivada/s hereda/n los elementos(atributtos y métodos) públicos y **protected**(no accesible desde fuera de la clase, pero lo heredan los hijos) de la clase base.

Sintaxis: la herencia se indica en la clase derivada(hija):

Class Person{

Public:

Person(char\* name):name(name){}

Protected:

Char\* name;

Private:

Printname(){ std::cout<<name<<std::endl;

}

Class Warrior: **public** Person{ //person es la clase base

Private:

Char\* weapon;

Public: (¿)

Warrior(char\* name, char\* weapon) : weapon(weapon ) , Person(name){}

}

Public: se quedan igual de visibles.

Protected: bajara un escalon de visibilidad de la clase padre. Public pasara a protected y protected pasa a privado.

Private: todo pasa a privado.

Declaración:

Person p(“Alex”);

Warrior w(“Eugenio”, “Humor”);

Las clases derivadas heredan todos los elementos public y protected excepto:

-Constructores\*

-Destructor\* \*No se heredan pero se llaman cuando creamos/eliminamos

Objetos de la clase derivada

-Operadores

-Friends

-----Ej2 templates ej herencia

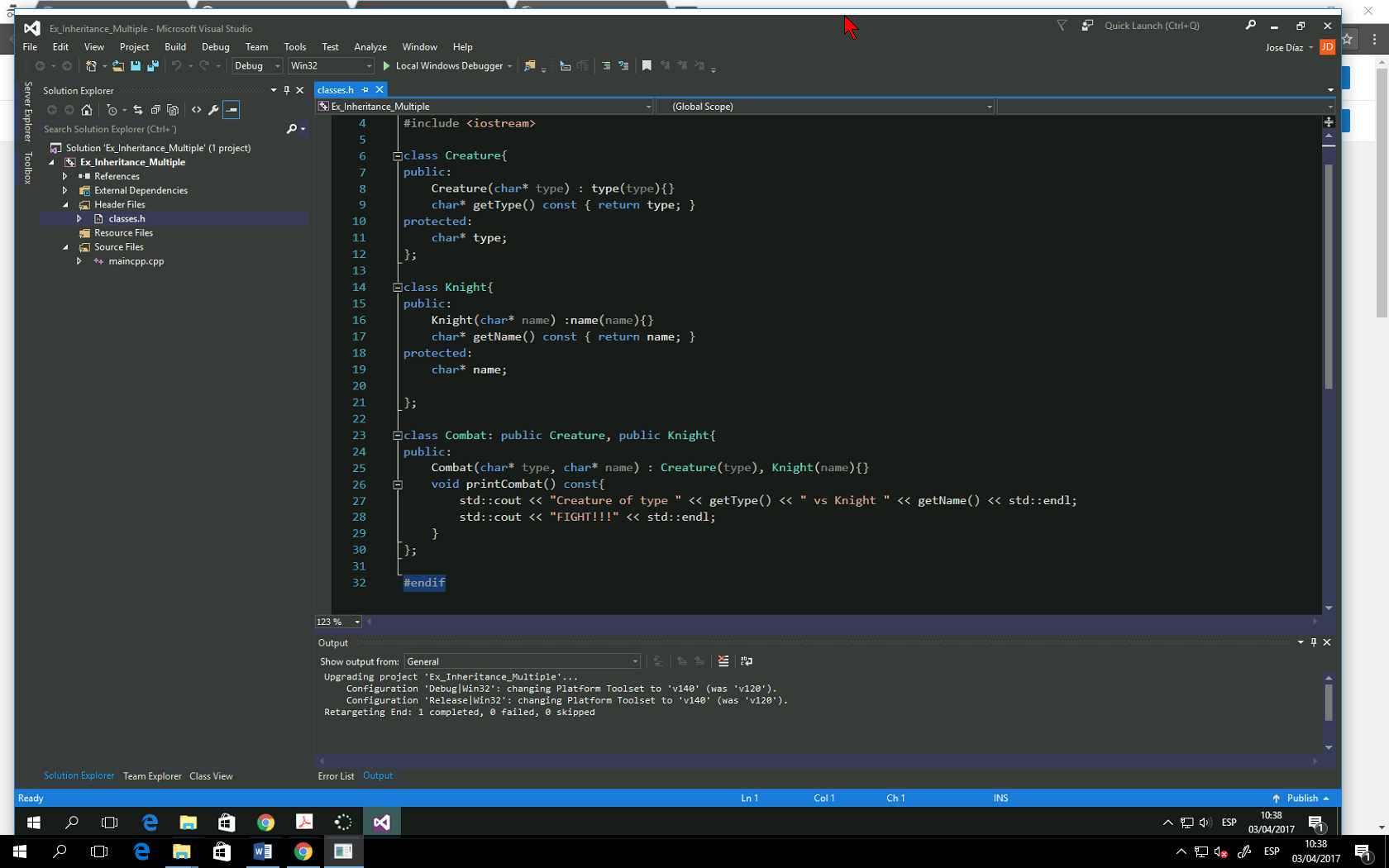
---03-04

**Herencia múltiple**

Lo que sea

Una clase hija hereda de uno o más padres.

Class nombre\_hijo : public nombre.padre1, public nombre padre2{…};



**Friendship C++**

Una función/acción(no devuelve nada, void) o una clase pueden ser declaradas como friend por otra clase, de manera que esa función/acción o clase tengan acceso a los elementos privados y protected de la clase.

\*Todo lo que declaro como friend puede ver lo que yo tengo, pero no al revés. La otra clase me tendría que declarar como friend también para yo poder verlo\*

Sintaxis: usando keyword friend

Friend Functions

Class Fighter{

Public:

Fighter(int life): life(life){}

Int getlife()const{return life;}

Friend void fatality(Fighter& f);

Private:

Int life;

};

Void fatility(Fighter &f){

f.life=0;}

Se incluye la declaración de la función en la clase precedida por friend.

Friendship con clases

Lo mismo que con funciones.

**IMPORTANTE:** la ‘friendship’ no es correspondida si no se especifica=>que una clase declare amiga a otra no significa que esta sea amiga de la primera.

**NO ES TRANSITIVA:** una clase no puede acceder a los miembros private y protected del amigo de un amigo.

Tenemos que ordenar las clases para que la que se declara como friend este antes que la que accede a esta, sino nos da error.

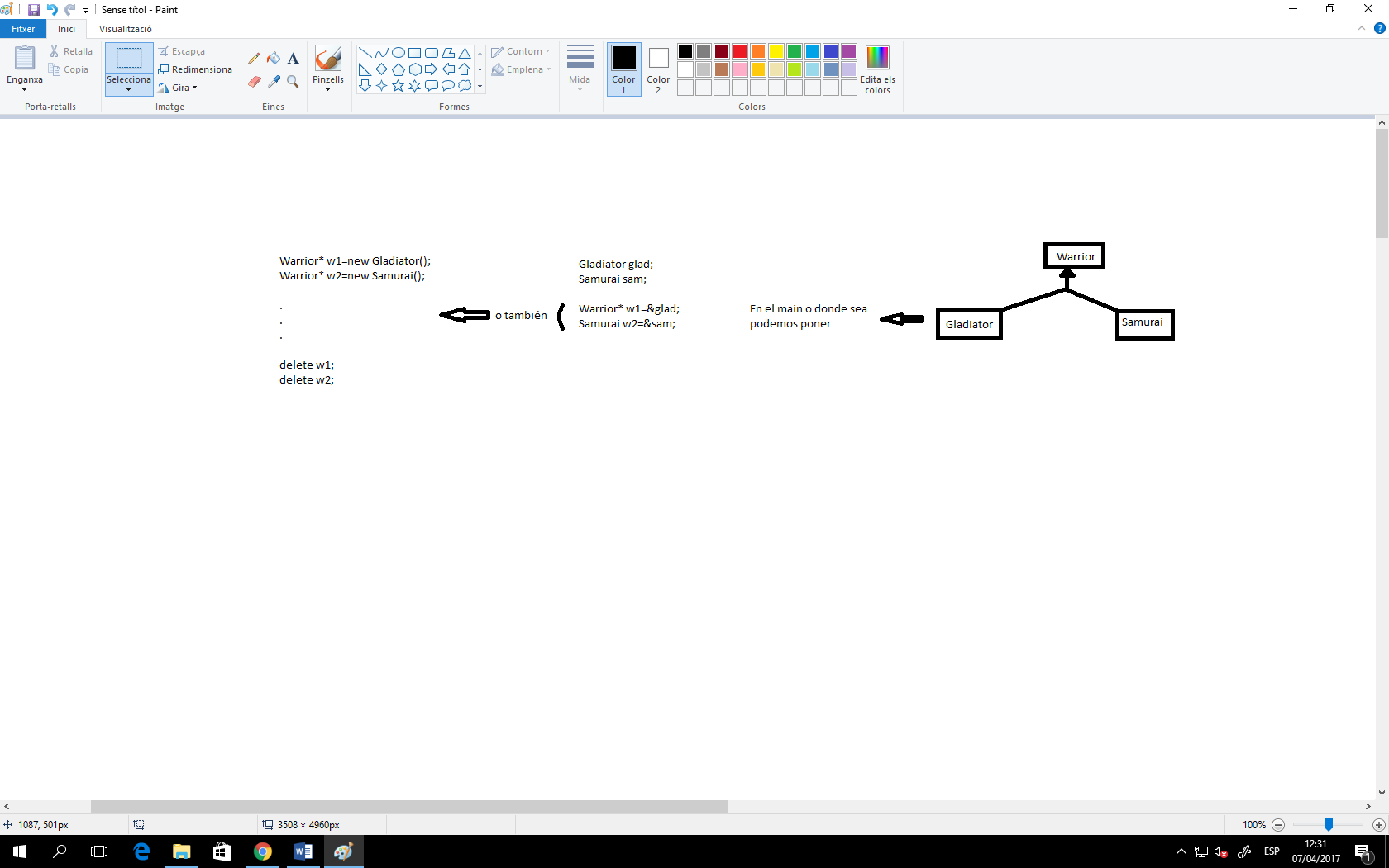
---ejercicio Friends --- jueves aula de estudio con Carlos

---07-04

**Polimorfismo**

Uno de los elementos clave de la herencia, es que un puntero de la clase derivada, es compatible con un puntero de la clase base.

El polimorfismo es sacarle partido a esto



Class Warrior{

Class Gladiator{

Public:

…

Int getLifeUnits() const;

…

Private:

Int lifeUnits;

};

Public:

…

Char\* getName() const;

…

Private.

Char\* name;

};

Int main(){

Warrior\* w=new Gladiator();

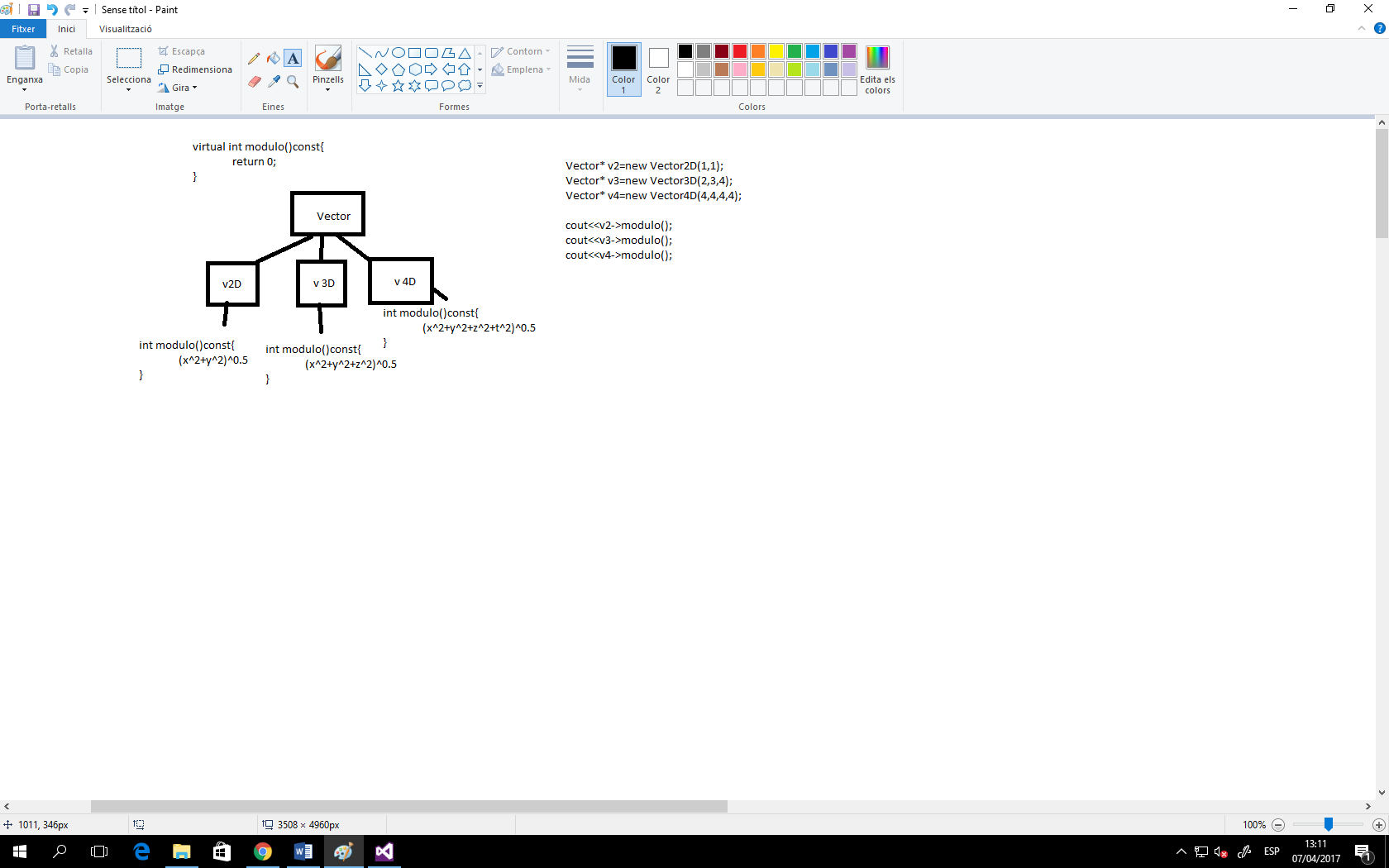
…

Delete w;

}

\*\*Mirar Prog2-07-04 para los virtuals\*\*

Virtual int modulo() const=0; //si ponemos esto envez de , es un virtual puro



Class abstract=> clase no instanciable

Caracteristicas de los métodos virtuales

-El constructor no puede ser virtual

-Los métodos virtuales se heredan

-En una clase con métodos virtuales, el destructor ha de ser virtual y se ha de reimplementar en las clases hijas.

Private:

Int\* vec;

Vec=new[4];

If(vec!=NULL){

Delete[]vec;

Vec==NULL;

}

--ej pokemon

https://mods.curse.com/addons/wow/pawn

<https://www.youtube.com/watch?v=CCreakh1wlM>

<https://mods.curse.com/addons/wow/atlasloot-enhanced>

<https://mods.curse.com/addons/wow/gtfo>

---21-04

DrMemory-> hago ejecutable (compilando) y lo arrastro a la aplicación de DrMem, y la aplicación lo ejecuta

---24-04

**Estructuras de datos**

Pila: estructura de datos que permite añadir/quitar elementos del “top”.

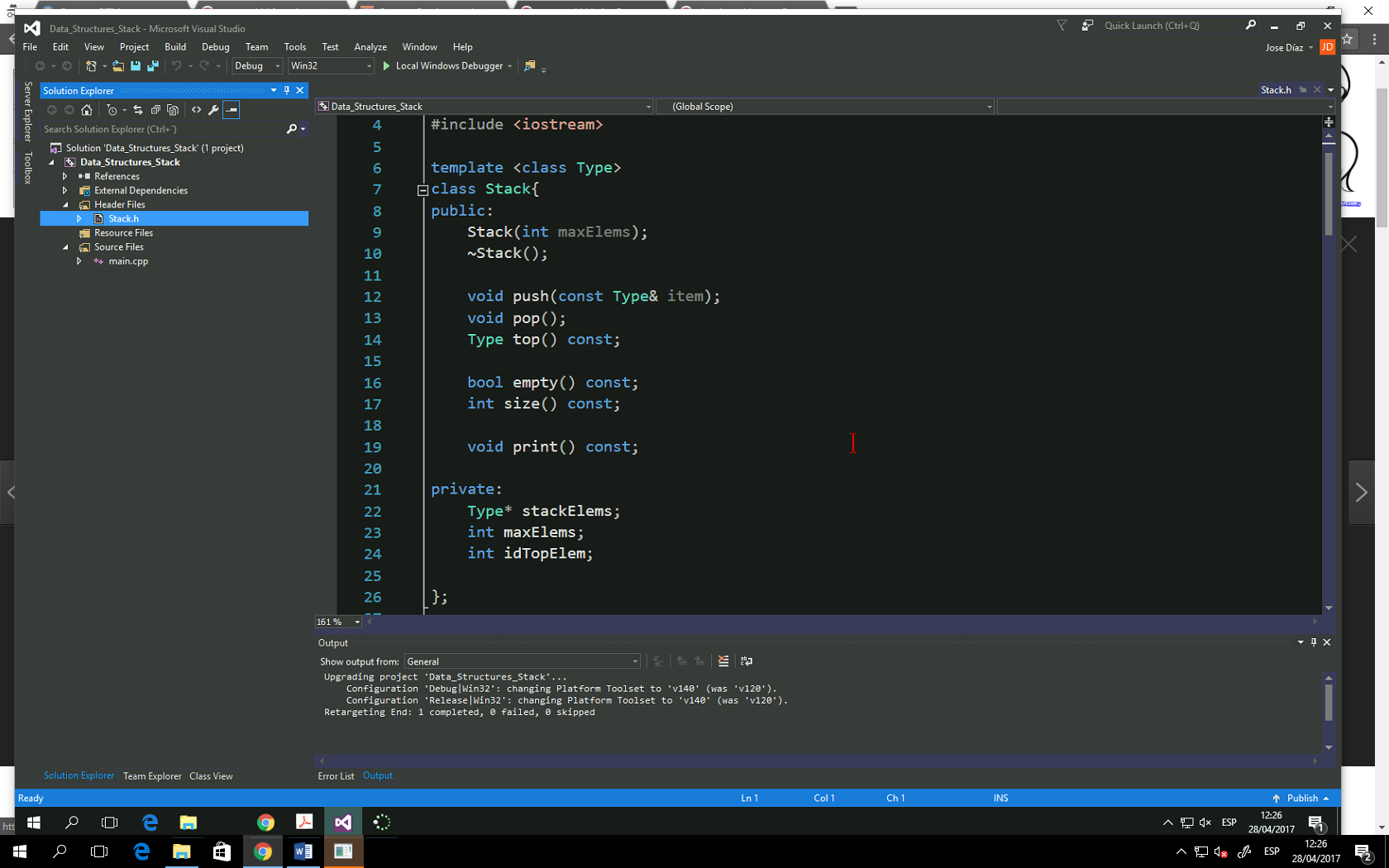
Tiene comportamiento “LIFO”: Last In, First Out.

Debe tener como mínimo los siguientes métodos:

-push -empty -pop -size -top

--Deberes ejercicio pila/stack

28-04---



**Cola-Queue**

Estructura de datos en la que se añaden elementos por detrás y se sacan por delante. Comportamiento FIFO: first in first out.

Métodos básicos:

-enqueue dequeue -size

Implementación: 2modos(implementado sobre array)

Lento: “dequeue” el primer elemento, “shiftar” el resto, este tendrá un solo índice, el elemento final, y “enqueue al final.

Rapido: guardar los ids deposición donde hacemos enqueue y dequeue y los actualizamos cuando añadimos o quitamos. \*OJO que el id puede rebasar el tamaño.\*

**Listas**

Secuencia finita de elementos donde añadimos o quitamos elementos en cualquier posición

Métodos básicos:

-pushFront(elem) insert(pos,elem) getValue(pos)

-pushBack(elem) remove(pos) empty()

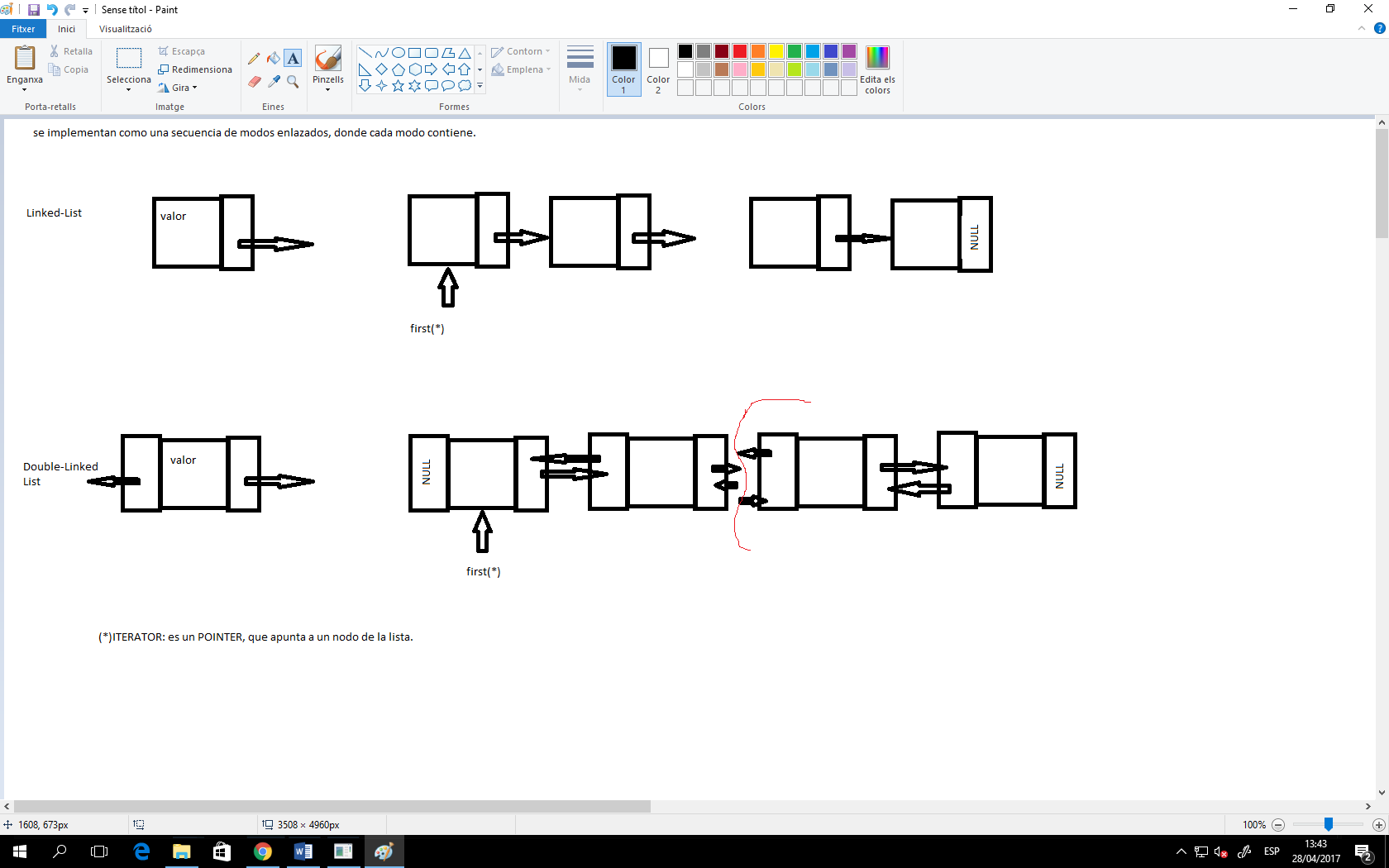
-popFront() front() size()

-popBack() back Clear()

Implementación:

Con un array de base->lento! Remove, insert… requieren shiftar!

**Linked** o **double-linked list** de **nodos**



Struct node{

Type value;

Node\* next;

}

---homework –hacer queue

---05-05

**String:**

Objeto que representa una cadena de caracteres. El último elemento tiene que ser “\0”.

Métodos:

-length() / size() (lo mismo; nº de letras del string sin contar \0).

-Empty() -reset()

-c\_str() (retorna el string) -print() (imprime en pantalla el string)

-operadores (==, !=, =, +=)

Implementación:

La única particularidad es que la gestión de memoria hemos de hacerla nosotros (cuando concatenamos strings, cuando destruimos).

**Importante:** cuando concatenamos strings por ejemplo, tenemos que alocatar memoria para el nuevo string, y preservar la información del antiguo.

Str1 str2 str1 str2

+ =

Preservar esto

Ej. Si tenemos “Hello” y “world” tendremos 2 strings de 6 caracteres (porque contamos el \0), pero al juntarlos solo tendremos 11 ya que uno de los \0 no nos servirá.

**Arrays dinámicos**

Todos los arrays son estáticos por naturaleza (determinado número de elementos en la declaración).

No podemos hacer int size=5; 🡺 tenemos que hacer

Float vec[size]; float vec[5];

En un array dinámico nosotros controlamos la memoria que se utiliza (aumentar disminuir depende del nº de elementos).

Podremos añadir o eliminar elementos de cualquier posición del array.

PROS: Acceso a cualquier posición inmediata.

CONS: añadir /eliminar elementos implica “shiftar” elems (costoso)

Métodos:

-insert(id,elem) -remove(id)

-operador[] -empty()

-size() -shrinkToFit() (ajusta la memoria utilizada por el array al num de elems)

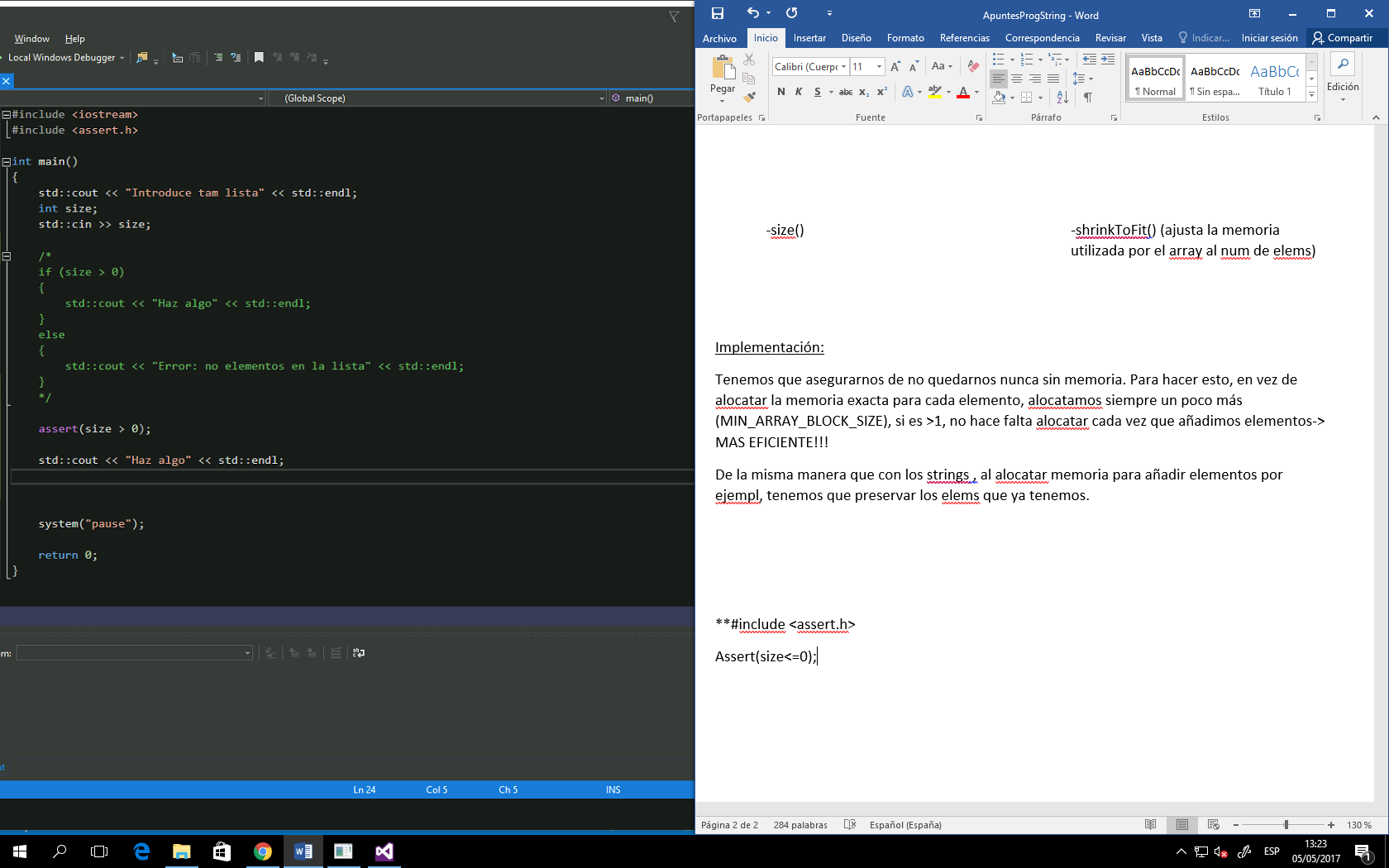
Implementación:

Tenemos que asegurarnos de no quedarnos nunca sin memoria. Para hacer esto, en vez de alocatar la memoria exacta para cada elemento, alocatamos siempre un poco más (MIN\_ARRAY\_BLOCK\_SIZE), si es >1, no hace falta alocatar cada vez que añadimos elementos-> MAS EFICIENTE!!!

De la misma manera que con los strings , al alocatar memoria para añadir elementos por ejempl, tenemos que preservar los elems que ya tenemos.

\*\*#include <assert.h>

Assert(size<=0); // si esto no se cumple, peta



\*\*lists

Insert:

1)Lista vacia->push Front(elem)

2)Lista con elems:

2.1)Insert al final->pushBack(elem)

2.2)Insert al principio->pushFront(elem)

2.3)Insert en posición intermedia-> a)crear nuedo nodo b)desplazarse hasta la posición c) modificar punteros d)comprobar el first

---08-05

**Recursividad**

Cuando una función se llama a sí misma para resolver el problema. Una función recursiva debe tener:

-Un (o varios) CASO BASE para el que sabemos la solución.

-Una forma de ir reduciendo nuestro problema en problemas más simples hasta llegar al base.

-Un CASO RECURSIVO que pasa “el problema simplificado” a la función.

PENSAMIENTO RECURSIVO ver la solución al problema como una versión más pequeña del problema. Imaginar que nuestra función resuelve el problema simplificado, y usarla para resolver el problema complejo.

¿Cómo hacerlo?

1)Identificar al CASO BASE el caso más simple para el que conocemos solución.

2)Retornar el valor del caso base.

3)CASO RECURSIVO llamar a la función pasándole como parámetro un “dato” (o conjunto de datos) más pequeño.

Implementación habitual:

If(caso base){return value:}

Else{return recursivefunction():}

---12-05

**Trees**

Representación jerárquica de los datos.

Al primer nodo se le llama ROOT (no tiene padre).

Cada “rama” es un SUBTREE.

Cada nodo sin ramas (o hijos) se le llama LEAF (no tiene hijos).

PRO: Si nodos ordenados, las búsquedas son muy rápidas!

Para recorrer el árbol se usa la RECURSIVIDAD=>cada subtree es un árbol más pequeño.

Tipos de tree:

Árboles binarios: cada nodo puede tener un máximo de 2 hijos.

Quadtrees: 4 hijos;

Octotrees: 8 hijos;

Métodos:

insertLeftChild(node,parentNode); clear();

insertRightChild( ll , ll ); empty();

remove(node); size();

search(node); getNumLevels();

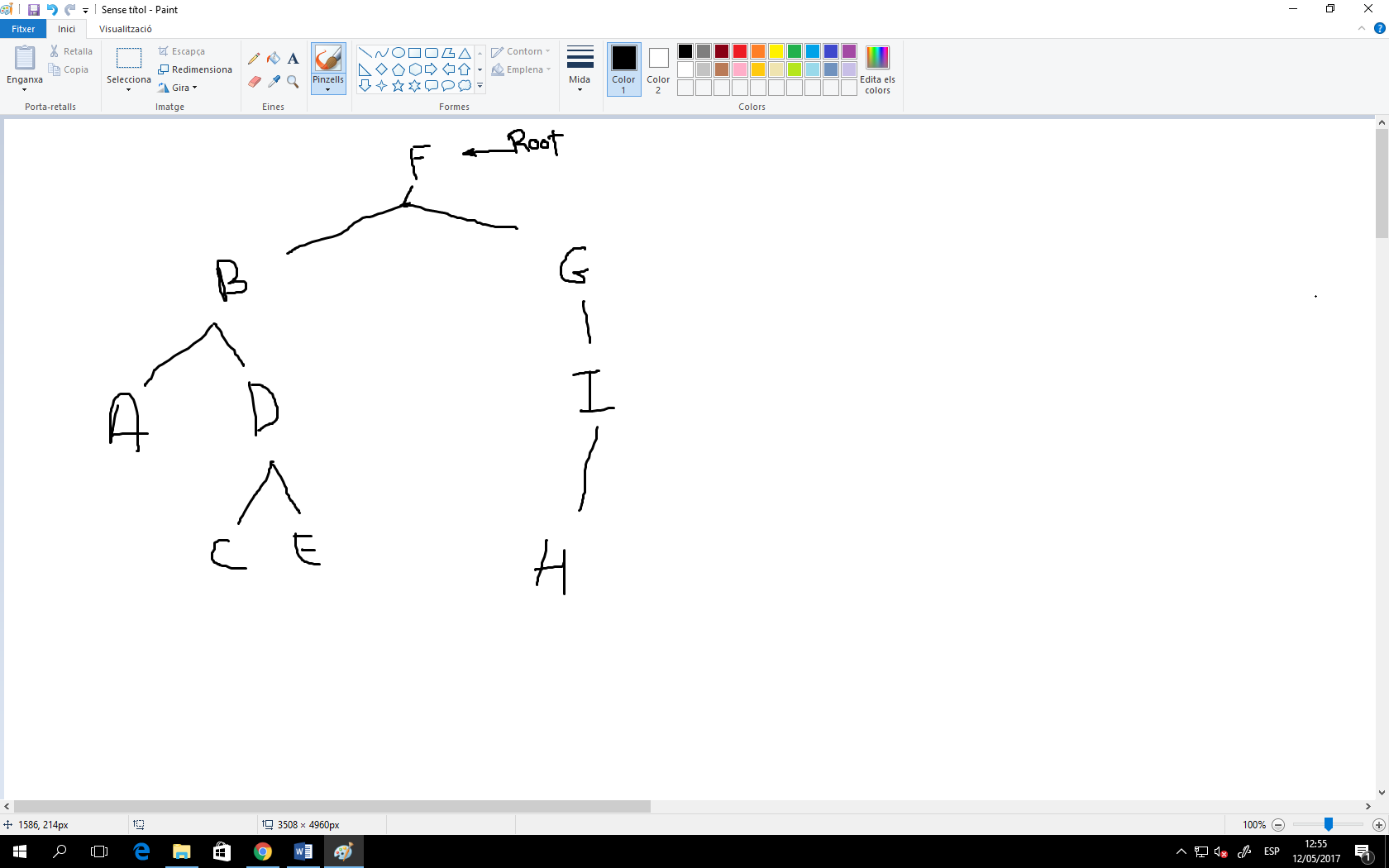
Recorridos en un árbol:

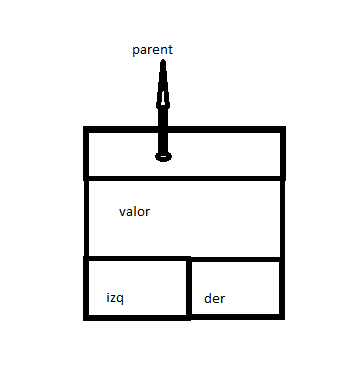
Preorden: nodo+hijo izq+hijo der. F,B,A,D,C,E,G,I,H

Inorden: hijo izq+nodo+hijo der. A,B,C,D,E,F,G,I,H

Postorden: hijo izq+hijo der+nodo. A,C,E,D,B,H,I,G,F

Breath-First : por niveles. F,B,G,A,D,I,C,E,H





Nodos de los tres. El root tiene el parent NULL y las leaves tienen hijos en NULL.

**Ordenación**

Bubble Sort: algoritmo más básico para ordenar arrays.

Basado en comparar elementos contiguos y hacer swap si elemi < o > elemi+1

Recorrer el array haciendo los swaps, hasta que en un recorrido no haga ningún swap.

(3,2,5,4) => (2,3,4,5) swaps=2 => (2,3,4,5) swaps=0 como que swaps es 0 entonces ya ha acabado.

\*\*\*https://www.reddit.com/r/manga/comments/3plu8t/recommend\_good\_manga\_with\_no\_anime\_adaptions/\*\*\*

**Game Development**

Al hacer una clase (en este ej vec3<TYPE>), no hace falta poner todas las variables en privado. Por ejemplo, la variable length que nos guarda la alargada del vec o lo que sea, tiene que ser privado ya que si nos la cambian desde fuera nos estropeara las funciones que creemos. Por otra parte la x y z no hace falta que sean privado.

Ex para próximo dia clase vec3 con operadores +, -, +=, =, ==, \*float, zeroc (poner todo a 0).

Operator + …..

Vec3<float> p1,p2;

P1.x=10; … .y=5 … .z=3;

Vec3<float> p3=p1+p2;

**Intro to XML**

---20-09

**Save and Load**

Serialize->pasar estado de una clase