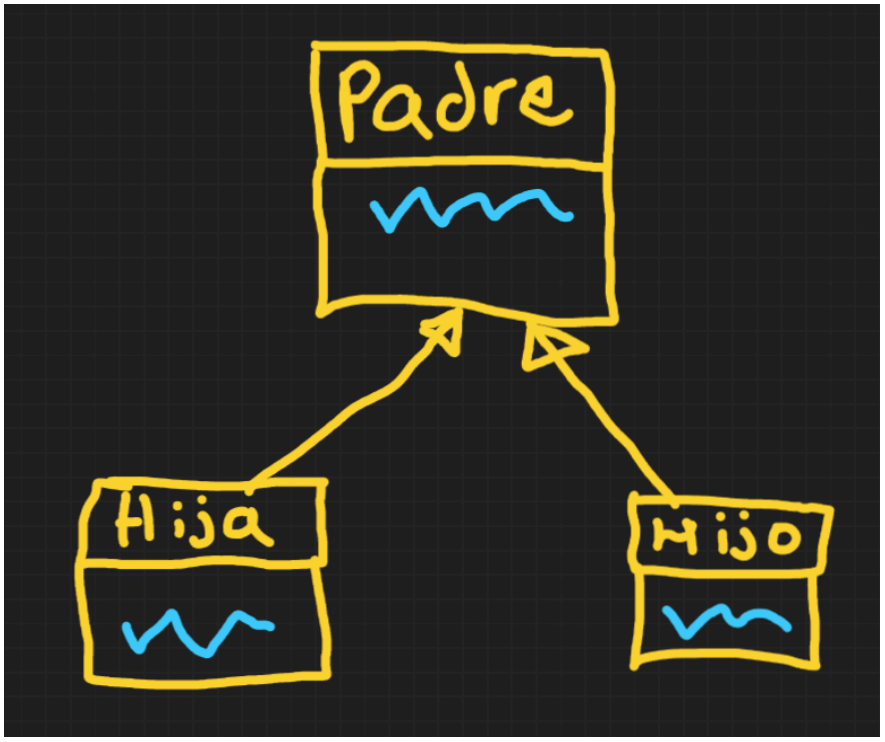
Vectores dinámicos:

Primero para usarlos precisamos de manejar punteros, recuerden que tienen mi video aca explicando cómo se usan, junto con detallitos que les pueden servir para entender mejor los vectores dinámicos.

Link: https://www.youtube.com/watch?v=1NnrYbQXfzA

Que son los punteros? | C++

Primero hay que entender una cosa, supongamos un esquema donde tenemos una clase **Padre**, abstracta, y 2 clases hijas: **Hijo e Hija**.



Entonces, siguiendo el esquema anterior, padre puede tener una serie de métodos, como por ejemplo **Hablar()**, donde al invocarlo diga *¨Hola¨*.
Hijo e Hija pueden tener el mismo método heredado del padre, pero sobrescrito, de tal forma que el Hijo puede decir: *“Chau”,* mientras que la hija diga *“Todo bien?”*.

```
class Padre{
public:
    virtual const char * Hablar(){
        return "Hola";
    }
};

class Hijo: public Padre{
    const char * Hablar() override{
        return "Chau";
    }
};

class Hija: public Padre{
    const char * Hablar() override{
        return "Todo bien?";
    }
};
```

¿Qué tiene que ver esto con los **Vectores Dinámicos**? Bastante de hecho.
Teniendo en cuenta esta estructura, y conociendo un poco de teoría de objetos, deberían saber que esto está acomodado perfectamente para hacer polimorfismo.
Es decir, desreferenciar un objeto de tipo puntero a Padre (*Padre * puntero = new Hijo/Hija/Padre*) y utilizar el método **Hablar(),** y encontrarnos que depende el tipo que instanciamos (que ponemos en el new), la respuesta será distinta, ya que cada clase implementa el método hablar de forma distinta o con resultados distintos.

Por lo tanto, deben saber como ya se ha dicho, que C++ implementa el polimorfismo mediante punteros, es decir, no es posible hacer una variable tipo Padre que al llamar sus métodos se comporte como Hijo o Hija.
Esto es clave, ya que entendiendo cómo funcionan los punteros, nos daremos cuenta que para manejar varias clases heredadas de **Padre,** y aprovechar el polimorfismo para no tener que andar usando el método **Hablar()** fijándonos de que tipo es cada clase, no podemos usar un solo puntero para manejar un array de estas clases, como veníamos trabajando hasta ahora con los *vectores dinámicos de tipo int, float, structs* o cosas asi.

```
Padre * puntero = new Padre[50]; // MAL
Padre ** putero = new Padre*[50]; // BIEEEN
```

Debemos usar dobles punteros, es decir **Padre ** puntero;,** y si alguno se pregunta porqué, la explicación rápida es así: Los punteros tienen una relación que podríamos decir matemáticamente, como que va de N a N-1.
Si hago un **int * puntero**, estoy diciéndole a “la computadora” que estoy creando un **puntero a entero**, por lo tanto al solicitar memoria mediante el puntero, como está especializado a apuntar a un **entero**, este solo nos permitirá crear clases de tipo **int.**
Esto para variables normales o structs no es drama, pero para clases si, ya que al aplicar el mismo ejemplo, diciendo que estoy haciendo un **puntero a Padre** (*ver el ejemplo de arriba*), fijense que estoy instanciando 50 veces la clase Padre. No a sus hijas, únicamente al Padre.
Y es aquí donde viene el problema, ya que de esta forma no estoy dando margen a guardar ahí las clases hijas, ya que el array es de un mismo tipo de dato común que es Padre.
Para poder guardar las clases hijas, necesito punteros a padre, no clases padre.

```
Padre * ejemplo = new Hijo;
```

Al usar un doble puntero, podríamos decir que estoy creando un **puntero que apunta a un puntero que apunta al padre.**
Siguiendo esta relación, al pedir memoria usando un doble puntero, lo único que puedo solicitar son **punteros que apuntan al padre**:

```
Padre ** ejemploMAL = new Padre;    a value of type "Padre *" cannot be used to initialize ar
Padre ** ejemploBIEN = new Padre*;
```

Notese que estoy pidiendo Padre, osea **puntero a padre.***

Esto me viene genial, ya que asi puedo pedir un **arreglo de punteros** de la clase base que necesite, y dentro de ese arreglo de punteros, puedo guardar mediante **new** las clases hijas de esta, o la clase base también si quiero y es posible, no hay impedimentos.

```
Padre ** ejemploBIEN = new Padre* [50];
ejemploBIEN[30] = new Hijo;
ejemploBIEN[25] = new Hija;
```

Mediante esto puedo emplear finalmente el polimorfismo de cada uno de los objetos que contenga en el array, haciendo por ejemplo que pueda recorrer el array que contiene los objetos, invocando el método de la clase base (que en este caso es **Hablar()**) y que depende como lo haya implementado cada clase, haga una cosa u otra.

Este es el resumen de cómo se manejan los arrays de objetos, y porque se deben hacer de esta manera. Hay muchas otras cosas más, pero esto es lo esencial para trabajar en el parcial.

Por último voy a explicar la función para modificar un vector dinámico:

```
// Version 1:
// Recibe por parametro un doble puntero a una clase, y un maximo.
// Amplia el array x2.
template <class Clase>
void AmpliarVector(Clase ** &arrayViejo, int &max)
{
    // Aumentamos x2 el maximo del array.
    int newMax = max*2;
    // Creamos un nuevo array de punteros, pero con el doble de tamanio.
    Clase ** newArray = new Clase*[newMax];
    // Copiamos los datos del array viejo dentro del array nuevo
    for (int i = 0; i < max; i++)
        newArray[i] = arrayViejo[i];
    // Liberamos la memoria del array viejo
    delete[] arrayViejo;
    // Actualizamos el max, y el array viejo
    arrayViejo = newArray;
    max = newMax;
}
```

La función es **sencilla**, en este caso la arme con template, para que cambien el nombre de Clase por el nombre de la clase que empleen **ustedes**.
Importante, si usan esta función con la template, al usarla por primera vez, en el parámetro de tipo deben poner **solo** el nombre de la clase, ya que sino al crearse la funcion hará literalmente un cuádruple puntero.

Ejemplo:
AmpliarVector<Padre>(arrayPadre, maxPadre);
El compilador al ver esto, reemplazará todos los nombres “Clase” que encuentre con Padre, formando asi un doble puntero de forma automática, ya que por ejemplo el prototipo de la funcion quedaria asi:
void AmpliarVector(**Padre** ** &arrayViejo, int &max)

Si colocamos Padre** en el parámetro de tipo, como se pueden dar cuenta, en vez de generar un Padre**, nos generaría un Padre****, entonces no andaría nada.

Volviendo a la explicación, esta función recibe un doble puntero, que apunta a un array de punteros, junto a un max que debería ser el máximo de ese vector.
Como dicen los comentarios, multiplica por dos el máximo, y crea un vector con el doble de espacio. Luego copia 1 a 1 los punteros del vector original en el nuevo, y al terminar, borra el viejo y le asigna el array nuevo.
Como pueden ver no es complicado, después hay otras versiones de esta función, sientanse libres de usar las que les guste.