Volem implementar una classe Vector que permeti fer una gestió intel·ligent d’un array dinàmic. Un objecte de la classe Vector ha de poder guardar un array dinàmic afegint alguns controls i precaucions en l’ús dels índexs per accedir als elements de l’array i en la gestió de la memòria dinàmica.

La classe Vector s’ha de declarar utilitzant templates per poder-la utilitzar amb qualsevol tipus de dades. El template tindrà un únic paràmetre que serà el tipus dels valors que es volen guardar a l’array.

Com a atributs, la classe vector haurà de tenir un array dinàmic del tipus del template i a més a més, tots els altres atributs que facin falta per poder implementar els mètodes que us indiquem a continuació:

1. Un constructor per defecte que inicialitza tots els atributs de forma segura. Inicialment l’array ha de tenir longitud 0.
2. El destructor, per alliberar la memòria de l’array.
3. Un mètode que permeti canviar el tamany de l’array dinàmic. S’haurà de reservar memòria per un nou array amb el nou tamany, fer la còpia de tots els elements de l’array antic al nou i alliberar l’array original. Si el nou array és més petit que l’original, es perden tots els elements que no caben en el nou array. Si el nou array és més gran, les posicions noves queden buides.

void redimensionar(int tamany);

1. Un mètode per recuperar la longitud de l’array:

int longitud();

1. L’operador [] per poder accedir a un element qualsevol de l’array. Si s’intenta accedir a un element fora dels índex vàlids de l’array s’ha de mostrar per pantalla un missatge d’error i retornar el valor per defecte del tipus que es guardi a l’array (obtingut a partir del constructor per defecte). Tingueu en compte que com que s’ha de retornar per referència, el valor per defecte haurà de correspondre al valor

T& operator[](int nIndex);

1. Un constructor de còpia. Tingueu en compte que copiar un objecte de la classe vector implica també copiar tots els elements del vector original al nou.
2. L’operador d’assignació, que ha de tenir en compte el mateix que s’ha comentat pel constructor de còpia.

Vector<T>& operator = (const Vector<T>& v);

1. Un mètode sort que permeti ordenar tots els elements del vector de més petit a més gran. Per ordenar un vector podeu utilitzar el mètode d’ordenació per intercanvi (o algorisme de la bombolla). Aquest algorisme va fent recorreguts de tots els elements del vector comparant cada element amb el següent i si és més gran els intercanvia de posició perquè quedin ben ordenats. Si es fa un nº de recorreguts del vector igual al nº d’elements, el vector queda completament ordenat. Podeu trobar una explicació més detallada d’aquest mètode i de com implementar-lo en aquest enllaç: <https://ca.wikipedia.org/wiki/Bubble-sort>

Per avaluar aquest exercici us donarem un programa principal que utilitza el template de la classe Vector amb dos tipus diferents: números enters i una classe Persona que us donem implementada i que guarda simplement el nom i l’edat d’una persona:

class Persona

{

public:

Persona();

Persona(const string &nom, int edat) : m\_nom(nom), m\_edat(edat) {}

~Persona();

string getNom() const { return m\_nom; }

int getEdat() const { return m\_edat; }

void setNom(const string &nom) { m\_nom = nom; }

void setEdat() { m\_edat = edat; }

private:

string m\_nom;

int m\_edat;

};

Tingueu en compte que per poder utilitzar el template Vector amb aquesta classe Persona, és necessari que la classe Persona tingui implementats l’operador de comparació >, l’operador de comparació == i el d’assignació =, de forma similar a com ho heu fet en exercicis anteriors. Aquests operadors no estan implementats a la versió que us donem nosaltres i per tant, haureu de modificar la classe Persona per afegir-los. En la comparació per saber si una persona és més gran que una altra, hem de comparar primer per l’edat i si les edats són iguals, llavors s’ha de comparar utilitzant el nom.

**Ejercicio 3**

En este ejercicio se va a escribir una clase de plantilla que haga de puntero inteligente. Un puntero

inteligente debería ser una clase que tuviese prácticamente la misma sintaxis que un puntero

normal, pero añada un manejo automático de la memoria para evitar problemas. En una primera

versión nuestra clase Puntero debería permitir código de este estilo:

// crear un puntero con valor inicial nulo.

Puntero<int> p1;

// crear un puntero inicializado con un puntero.

Puntero<int> p2 = new int(10);

// asignar un puntero a un puntero nulo, tomar posesión del puntero

p1 = new int(77);

// asignar un puntero nuevo a un puntero con valor previo,

// la memoria previa se libera y se toma posesión de la nueva

p2 = new int(45);

// asignación entre punteros inteligentes, el puntero destino libera cualquier memoria que pudiera

poseer, toma la posesión del puntero origen y lo deja a nulo.

// Dereferenciación del puntero

std::cout << \*p1 << ' ' << \*p2 << std::endl;

// Acceso a miembros para punteros a clases

struct A

{

void f();

};Puntero<A> p\_a = new A;

p\_a->f();

// Al final del bloque se destruyen p1, p2 y pa, liberando la memoria a la que apuntan.

Así lo que se exige al puntero inteligente sería lo siguiente:

1. Un constructor por defecto, uno de copia y un operador de asignación (¡cuidado con la

autoasignación!)

2. Un destructor

3. Que cuando se intente acceder a un puntero nulo se imprima un mensaje de error antes de

que ocurra la catástrofe inminente (en código real en vez de imprimir un mensaje se usaría

un assert o una excepción como se explicará en una lección posterior).

4. Una sobrecarga del operador\* y del operador->

Escribir un programa que pruebe diversas cosas, se puede partir del ejemplo de arriba.

Nuestro puntero anterior existe también ya en la STL, denominado auto\_ptr y dentro de la cabecera

<memory>. La característica más importante de este puntero es que es los punteros almacenados

son exclusivos (al igual que en nuestro puntero)

Cuestiones adicionales:

1. \* ¿Valdría la clase anterior como puntero inteligente para arrays? (Hay varias razones)

2. \*\* ¿Sería posible escribir una clase sencilla que valiese para arrays y punteros normales?

3. \*\*\* Nuestro puntero es exclusivo. Sin embargo en la mayor parte de los casos interesa que

cada variable puntero asignada desde un mismo puntero original apunte al mismo dato sin

que ninguna pierda la referencia, ya que así es como funciona un puntero real. Una manera

de llevar a cabo este propósito es llevar una cuenta del número de referencias al puntero y

sólo liberarlo cuando esta llega a cero. En este apartado se trata de modificar nuestro puntero

para incluir el contador de referencias (Pista: el contador de referencias tiene que ser

compartido también por todas las variables que apuntan al mismo dato).

Notas:

Un puntero de este estilo no es la solución definitiva a todos los problemas de manejo de

memoria. Considerar que pasa por ejemplo si tenemos dos objetos con sendos punteros que

se apuntan recíprocamente.

Por desgracia, en la STL no existe ningún puntero que tenga esta funcionalidad. Para los

curiosos, sí puede encontrar una clase shared\_ptr como parte