Ejercicio 1

Definir e implementar una clase Punto, que posea los elementos públicos y privados:

Variables miembro privadas:

mx y **my** - del tipo **double**, que representan las coordenadas (x; y) del punto.

Métodos miembro públicos:

setPunto: recibe los valores de x e y en dos variables double.
getPunto: devuelve el valor del punto (en formato class Punto). En formato inline
setX y **setY**; que dan valor a x e y. En formato inline
getX y **getY**; que devuelven los valores de x e y. En formato inline

La creación del objeto debe permitir o no, las siguientes expresiones:

Punto pa; // mx y my se inicializan en cero.
Punto pb (23.3,56.8); // mx se inicializa con 23.3 y my con 56.8.

La inicialización del objeto con un solo parámetro debe dar error de sintaxis, por ejemplo la línea:

Punto pc (34.4); // error de sintaxis.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(3000.12,4.45);
 Punto r;
 // Punto q(5.7); -- Error de sintaxis
 cout <<"1. punto p: ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 cout <<"2. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 r.setX(-2000.22);
 r.setY(3.33);
 cout <<"3. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 p.setPunto(9900.9,8800.8);
 cout <<"4. punto p: ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 r=p.getPunto();
 cout <<"5. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 PRESS_KEY;
}
```

Ejercicio 2

Modificar la clase del punto anterior, de manera de verificar que los valores de x;y se encuentren siempre dentro del rango -1000 a 1000, caso contrario su valor debe limitarse a estos valores.
Este control se debe realizar a través de miembros privados de la clase.

Ejercicio 3

Modificar el punto anterior de manera que acepte también el Método set_punto (pa); en donde pa es un objeto Punto y acepte la creación del objeto de manera que la expresión Punto pc (34.4); sea aceptada asignando el valor pasado como parámetro a la coordenada x y asigne el valor cero a la coordenada y.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(1234.56);
 Punto r(12,34);
 cout <<"1. punto p: ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 cout <<"2. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 p.setY(3.33);
 r.setPunto(p);
 cout <<"3. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 PRESS_KEY;
}
```

Ejercicio 4

Modificar el punto anterior de manera que acepte la creación de un objeto de la siguiente manera:

Punto pd (pc); o Punto pd=pc; // donde pc es un objeto Punto.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(12.34,-56.78);
 Punto r(p);
 Punto q=p;
 cout <<"1. punto p: ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 cout <<"2. punto r: ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 cout <<"3. punto q: ("<<q.getX()<<";"<<q.getY()<<")"<<endl;
 PRESS_KEY;
}
```

Ejercicio 5

Modificar el punto anterior para permitir la suma (+), resta (-) y asignación (=) de objetos tipo Punto.
Nota: tener en cuenta que las operaciones de deben controlar que los valores de x e y, no desborden el rango de +/- 1000.
No utilizar friend, para realizar la sobrecarga de los operadores suma (+) y resta (-).

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(12.34,-56.78);
 Punto r(1,4);
 Punto q;
 cout <<"1. punto p= ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 cout <<"2. punto r= ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 q=p+r;
 cout <<"3. punto p+r: q= ("<<q.getX()<<";"<<q.getY()<<")"<<endl;
 q=p-r;
 cout <<"4. punto p-r: q= ("<<q.getX()<<";"<<q.getY()<<")"<<endl;
 Punto s(990,-990);
 cout <<"5. punto s= ("<<s.getX()<<";"<<s.getY()<<")"<<endl;
```

```
 q=s+p;
 cout <<"6. punto s+p: q= ("<<q.getX()<<";"<<q.getY()<<")"<<endl;
 q=r+47;
 cout <<"7. punto r+47: q= ("<<q.getX()<<";"<<q.getY()<<")"<<endl;
 PRESS_KEY;
}
```

Preguntas:
- ¿Requiere sobrecargar el operador de asignación (=)? ¿Por qué?
- Justifique la ejecución de la línea q=r+47; (como se produce la suma entre el objeto Punto y el objeto int, suponiendo que mantuvo la consigna de escribir lo menos posible y no sobrecargó la suma de Punto con int)

Ejercicio 6

Modificar el punto anterior para permitir la ejecución del siguiente código. Recordar realizar siempre la menor cantidad de código. En este caso, sería que la implementación de la suma y resta se realice cada una con un solo método miembro.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(12.34,-56.78);
 Punto r,s;
 s=78+p;
 r=78-p;
 cout <<"1. punto p= ("<<p.getX()<<";"<<p.getY()<<")"<<endl;
 cout <<"2. punto 78+p: s= ("<<s.getX()<<";"<<s.getY()<<")"<<endl;
 cout <<"3. punto 78-p: r= ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 r=p+s-45;
 cout <<"4. punto p+s-45: r= ("<<r.getX()<<";"<<r.getY()<<")"<<endl;
 PRESS_KEY;
}
```

Ejercicio 7

Modificar el punto anterior para permitir la comparación entre objetos tipo Punto: igualdad (==); desigualdad (!=), mayor (>) y menor (<), sobrecargar el flujo de salida y entrada (<< y >>).
En el caso de la sobrecarga de mayor (>) y menor (<), se debe comparar la distancia que existe al centro de coordenadas (0;0).

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto p(12.34,-56.78);
 double r=45;
 cout <<"1. punto p= "<<p<<endl;
 cout <<"2. punto r= "<<r<<endl;
 cout <<"Ingrese valor del punto"<<endl;
 Punto h;
 cin >>h;
 cout <<"3. punto h= "<<h<< "(*)" <<endl<<endl<<endl;
 cout <<" 4. Es h igual a p ? : "<<((h==p)?"si":"no")<< endl;
 cout <<" 5. Es h distinto a p ? : "<<((h!=p)?"si":"no")<< endl;
 cout <<" 6. Es h mayor a p ? : "<<((h>p)?"si":"no") << endl;
 cout <<" 7. Es h menor a p ? : "<<((h<p)?"si":"no") << endl<<endl;
 cout <<" 8. Es h igual a r ? : "<<((h==r)?"si":"no")<< endl;
 cout <<" 9. Es h distinto a r ? : "<<((h!=r)?"si":"no")<< endl;
 cout <<"10. Es h mayor a r ? : "<<((h>r)?"si":"no") << endl;
 cout <<"11. Es h menor a r ? : "<<((h<r)?"si":"no") << endl<<endl;
 PRESS_KEY;
}
```

Ejercicio 8

Agregar a la clase Punto, del ejercicio anterior, dos métodos públicos del tipo inline que indiquen la cantidad de instancias del tipo objeto que han sido creadas y cuantas se encuentran en memoria.
Nota: los contadores debe ser miembros privados de la clase.
Justificar la invocación de los métodos getCantCreada y getCantExistente antes de la creación de cualquier objeto.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

void ff (void)
{
 Punto p,q,w;
 Punto h(34);
 Punto r=h;
 cout << "a. Puntos Creados:" << Punto::getCantCreada()
      << " - Existentes:" << r.getCantExistente() << endl;
}
 int main(int argc, char *argv[])
{
 cout << "1. Puntos Creados:" << Punto::getCantCreada()
      << " - Existentes:"<< Punto::getCantExistente() << endl;
 Punto p(12.34,-56.78);
 cout << "2. Puntos Creados:" << p.getCantCreada()
      << " - Existentes:" << p.getCantExistente() << endl;
 Punto h(p);
 cout << "3. Puntos Creados:" << Punto::getCantCreada()
      << " - Existentes:" << Punto::getCantExistente() << endl;
 ff();
 cout << "4. Puntos Creados:" << Punto::getCantCreada()
      << " - Existentes:" << Punto::getCantExistente() << endl;
 PRESS_KEY;
}
```

Ejercicio 9

Agregar un método público “set_limites (float,float)”, que modifique el rango válido de x e y de la clase.
Esto significa que el rango de todos los objetos existentes y de los objetos por crear se verán afectados por este método.
Este método no debe modificar los valores de coordenadas x;y. (No importa que éstos queden fuera de rango).
El primer parámetro corresponde al límite inferior y el segundo al superior. Si el límite inferior no es menor al superior, se debe omitir el cambio del rango.
Implementar además las funciones “getLimiteSup” y “getLimiteInf”, del tipo inline, para saber cuáles son los valores de estos límites. Por omisión, los límites son +/- 1000.

Ejemplos:

Punto **Pa**(5000,-5000); toma el valor (1000;-1000) por los límites definidos por omisión.

Si luego ejecuto:

Punto::set_limites (50,-50);

Punto **Pb**(5000,-5000); toma el valor (50;-50) por los límites definidos explícitamente.

Pa mantendrá el valor (1000;-1000), sin embargo si ejecuto Pa=Pa+100, su nuevo valor será (50;-50) pues se ve afectado por el nuevo rango definido.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 cout <<"1. Rango de punto: "<<Punto::getLimiteInf()<<
":"<<Punto::getLimiteSup()<<endl;
 Punto p(3000.12,5000);
 Punto r(12.34,34.56);
 cout <<"2. punto p: "<<p<<endl;
 cout <<"3. punto r: "<<r<<endl;
 Punto::setLimites(50,85);
 cout <<"4. Rango de punto: "<<p.getLimiteInf()<<
":"<<p.getLimiteSup()<<endl;
 cout <<"5. punto p: "<<p<<endl;
 cout <<"6. punto r: "<<r<<endl;
 Punto t;
 cout <<"7. nuevo punto t: "<<t<<endl;
 r=p; // como la igualdad no esta redefinida, no se ve afectada por el
nuevo límite
 cout <<"8. r=p r: "<<r<<endl;
 r.setPunto(p);
 cout <<"9. setPunto r: "<<r<<endl;
 r.setLimites(500,-85);
 cout <<"10. Rango de punto: "<<Punto::getLimiteInf()<<
":"<<Punto::getLimiteSup()<<endl;
 PRESS_KEY;
}
```

Nótese que al crear el objeto t, con parámetros por omisión, debería haber tomado el valor (0;0), sin embargo como el límite inferior es '50', toma el valor (50;50);
Además, como el operador igualdad (=) no está sobrecargado, la operación r=p, será una copia bit a bit y no se controlará el rango, por ende r poseerá valores fuera del rango.

Pregunta: ¿por qué no se pude definir las funciones getLimiteSup y getLimiteInf como const?

Ejercicio 10

Sobrecargar el pre y post incrementó de manera que incremente en una unidad tanto el valor de x como de y.

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto r(12.34,34.56);
 cout <<"1. punto r: "<<r<<endl;
 cout <<"2. punto r++: "<<r++<<endl;
 cout <<"3. punto r: "<<r<<endl;
 cout <<"4. punto ++r: "<<++r<<endl;
 PRESS_KEY;
}
```

Ejercicio 11

Modificar el punto anterior de manera que acepte los operadores new y delete.
Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 Punto *r=new Punto(12.34,34.56);
 cout << "1. punto r: " << *r << endl;
 cout << "2. Puntos Creados:" << r->getCantCreada()
      << " - Existentes:" << Punto::getCantExistente() << endl;
```

```
 delete (r);
 cout << "3. Puntos Creados:" << Punto::getCantCreada()
      << " - Existentes:" << Punto::getCantExistente() << endl;
 PRESS_KEY;
}
```

¿Cuáles fueron las modificaciones necesarias? Justificar.

Ejercicio 12

Se debe implementar la clase IntArr, la cual es una clase para trabajar con una array o vector de datos tipo int en forma dinámica, cuya definición es la que sigue:

```
class IntArr
{
 private:
 int * p;
 int size;
 int used;
 public:
 IntArr (int sz);
 IntArr (int sz,int qtty,int *vec);
 ~IntArr();
 void prtArr (void) const;
};
```

En donde

p: es el puntero al array dinámico.
size: es la cantidad de elementos del array p.
used: es la cantidad de elementos usados del array p. used será siempre <= a size.
IntArr (int sz): crea un elemento IntArr con sz elementos disponibles.
IntArr (int sz,int qtty, int *vec): crea un elemento IntArr con sz elementos disponibles, y copia qtty elementos del array vec al array p.
~IntArr (int sz): destructor.
prtArr: imprime el Array en pantalla.

Como ejemplo de aplicación se presenta el siguiente ejemplo con su respectiva salida:
Nota: si en IntArr (sz,qtty,vec); qtty es mayor a sz, se debe solucionar el error igualando sz a qtty.

Código Fuente:

```
using std::cout;
using std::endl;
```

```
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 IntArr A(30);
 int v_aux[]={23,4,54,634,6677,32,56};
 IntArr B(40,sizeof(v_aux)/sizeof(int),v_aux);
 A.prtArr();
 B.prtArr();
 PRESS_KEY;
}
```

Ejercicio 13

Agregar a la clase IntArr, los siguientes métodos públicos:
getSize del tipo inline que devuelve el tamaño del IntArr.
getUsed del tipo inline que devuelve la cantidad de elementos usados del IntArr.
getAvg devuelve el promedio de los elementos del IntArr. El promedio devuelto es del tipo double.
Y sobrecargar el método prtArr, de manera de indicar cuantos elementos imprimir desde el inicio del Array.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 IntArr A(30);
 int v_aux[]={23,4,54,634,6677,32,56};
 IntArr B(40,sizeof(v_aux)/sizeof(int),v_aux);
 A.prtArr();
 B.prtArr();
 B.prtArr(3);
 cout<<"\n\nObjeto B -"<<endl;
 cout<<" size:"<<B.getSize()<<endl;
 cout<<" used:"<<B.getUsed()<<endl;
 cout<<" promedio:"<<B.getAvg()<<endl;
 PRESS_KEY;
}
```

Ejercicio 14

Agregar a la clase IntArr, los siguientes métodos públicos:
addElement (int xx): el cual agrega el elemento xx al final del array “IntArr”.
addElement (int qtty, int *vec): el cual agrega los qtty elementos que se encuentran en vec al final del array “IntArr”.

Nota: tener en cuenta que al agregar elementos al objeto, se puede desbordar el Array dinámico, por lo tanto habrá que redimensionarlo. En tal caso y en forma preventiva solicitar espacio para 5 elementos más de los necesarios.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();

int main(int argc, char *argv[])
{
 int v_aux[]={0,5,10,15,20,25,30,35,40};
 IntArr A(10,sizeof(v_aux)/sizeof(int),v_aux);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 A.addElement(77);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 A.addElement(11);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 A.addElement(8,v_aux);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 PRESS_KEY;
}
```

Ejercicio 15

Modificar los métodos addElement, incluidos en el ejercicio anterior, de manera de poder indicar a partir de qué se deben agregar los nuevos elementos. Siendo el nuevo prototipo como sigue:

addElement (int pos, int xx).
addElement (int pos, int qtty, int *vec)

En donde pos, indica la posición a partir de donde comienza la inserción. Los demás parámetros mantienen su significado.

Nota: En caso que pos fuese negativo, se considera inserción desde el inicio y sí el valor de pos, supera la máxima ubicación posible dentro del Array dinámico, se debe insertar al final.

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();
#define SZ_VEC(x) (sizeof(x)/sizeof(x[0]))

int main(int argc, char *argv[])
{
 int v1[]={0,5,10,15,20,25,30,35,40};
 int v2[]={1,2,3,4,5,6};
 IntArr A(10,SZ_VEC(v1),v1);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 A.addElement(0,77);
 A.addElement(56,11);
 A.addElement(4,SZ_VEC(v2),v2);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 A.addElement(4,99);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 PRESS_KEY;
}
```

Ejercicio 16

Agregar a la clase IntArr del ejercicio anterior, los elementos necesarios para permitir las sentencias que se visualizan en el siguiente código:

Código Fuente:

```
using std::cout;
using std::endl;
#define PRESS_KEY std::cout<<"\nPresione Enter para continuar . .
.\n";std::cin.get();
#define SZ_VEC(x) (sizeof(x)/sizeof(x[0]))

int main(int argc, char *argv[])
{
 cout<<"\n==================> Inicio <==================\n";
 int v1[]={0,5,10,15,20,25,30,35,40};
```

```
 int v2[]={1,2,3,4,5,6};
 IntArr A(10,SZ_VEC(v1),v1);
 IntArr B(10,SZ_VEC(v2),v2);
 IntArr C=B;
 B.addElement(0,99);
 cout<<" size:"<<A.getSize()<<endl<<" used:"<<A.getUsed()<<endl;
 A.prtArr();
 cout<<"\n> Array B\n"<<B;
 cout<<"\n> Array C\n"<<C;
 cout<<"\n==================> Medio <==================\n";
 A=B+C;
 cout<<"\n> Array A=B+C\n"<<A;
 IntArr D(10,SZ_VEC(v1),v1);
 D=D;
 cout<<"\n> Array A\n"<<A;
 cout<<"\n==================> Medio <==================\n";
 D+=B;
 cout<<"\n> Array D+=B\n"<<D;
 PRESS_KEY;
}
```

Ejercicio 17

Se desea realizar una clase en c++ para utilizar en una aplicación de ventas para una máquina expendedora de ananás. Para eso, se tiene la siguiente definición de una clase fruta, que es genérica, de modo de poder ser utilizada en cualquier otra máquina expendedora (jugos exprimidos, de licuados, y de otras frutas, etc).
A saber:

```
class fruta
{
    private:
    char*;
    unsigned int;
    float;
    nombreFruta;
    pesoEnGramos;
    precioPorKilo;
    public:
    // Constructores
    fruta(); //Por defecto.
    fruta(const char*, unsigned int, float); //Parametrizado
    fruta(const fruta&); //Copia
    ~fruta();
    // Metodos
```

```
    void setNombre(const char*);
    char* getNombre (void);
    void setPeso(unsigned int);
    unsigned int getPeso(void);
    void setPrecioKilo(float);
    float getPrecioKilo(void);
    float getPrecioFinal(void);
    // Sobrecarga de operadores.
    friend ostream& operator<< (ostream& , const fruta& );
};
```

Se solicita implementar los siguientes constructores:

1. Constructor por defecto. Debe inicializar el nombre de la fruta a "No detallado", el peso de la fruta a 0, y el precio por kilogramo a 0.
2. Constructor parametrizado. Recibe el nombre de la fruta, el peso de la misma y el precio por kilo. Si el nombre de la fruta tiene mas de 32 caracteres, debe truncarlo en 32.
3. Constructor de copia. Copia los datos del objeto fruta pasado como argumento al objeto fruta que se está instanciando. En todos los casos, tiene que realizar correctamente el manejo de memoria dinámica. No asumir que hay memoria disponible. Chequear que el sistema operativo haya podido asignar la memoria solicitada.

Ejercicio 18

Realizar los siguientes métodos, cuyo comportamiento se puede deducir del nombre.

void setNombre (const char*) → Modifica el nombre de la fruta. Debe limitar a 32 caracteres.
char* getNombre (void) → Devuelve un puntero al nombre de la fruta.
void setPeso (unsigned int) → Modifica el peso de la fruta.
unsigned int getPeso (void) → Devuelve el peso de la fruta.
void setPrecioKilo (float) → Modifica el precio por kilo de esta fruta.
float getPrecioKilo (void) → Devuelve el precio por kilo de esta fruta.
float getPrecioFinal (void) → Devuelve el precio a cobrar (hace la cuenta).

Ejercicio 19

Implementar la sobrecarga del operador << de modo que al utilizarlo imprima en pantalla los datos del objeto que lo utiliza. Por ejemplo: si tenemos el siguiente fragmento de programa:
fruta A((const char*) "Anana", 1200, 250);
Se trataria de un Anana, que pesa 1,2 kilogramos, y como el precio por kilo es $250, su precio total sería $300. Con lo cual, al hacer

cout << A << endl;

El programa debería imprimir en pantalla:

Datos de la fruta:
Nombre: anana.
Peso [gramos]: 1200
Precio por kilo [$]: 250
Precio de esta unidad [$]: 300

Ejercicio 20

Modificar la clase fruta, agregando una variable estática privada que cuente la cantidad de objetos instanciados.
Modificar los constructores (todos ellos) de modo que incrementen estas variables cada vez que se instancia un nuevo objeto de tipo fruta. Incluir un método estático para poder mostrar en pantalla el valor de esta variable.
Implementar un main que muestre el uso de todos los items pedidos.

Ejercicio 21

Dada la clase DisplayLCD

```
class DisplayLCD
{
    private:
    unsigned int Renglones;
    unsigned int Caracteres_Por_Renglon;
    unsigned char* Datos;

    public:
    // Constructores
    DisplayLCD ();
    DisplayLCD (unsigned int renglones, unsigned int caracteres);
    // Metodos
    void DisplayLCD::SetRenglon(unsigned int renglon, unsigned char* texto);
    // Sobrecarga de operadores.
    DisplayLCD :: operator+(DisplayLCD a);
    friend ostream& operator << (ostream& o, const DisplayLCD d);
}
```

Que se utiliza para controlar un display LCD alfanumérico conectado a nuestro sistema embebido de una manera que en este momento no nos preocupa se pide:

- Implementar el constructor por defecto, que debe hacer que el objeto creado sin parámetros tenga 2 renglones, y 16 caracteres por renglón. Pedir memoria en forma dinámica para poder alojar mensajes de esa longitud.

- Implementar el constructor parametrizado. Pedir memoria en forma dinámica para poder alojar los mensajes de acuerdo a la cantidad de renglones y de caracteres por renglón del objeto que se esta creando.

- El método SetRenglon que coloque en el renglón pasado como 1er parámetro, el mensaje pasado como segundo argumento. Si el mensaje es mas largo de la cantidad de caracteres por renglón del objeto, truncarlo en el tamaño de renglón.

- Realizar la sobrecarga del operador +, de modo tal que se haga la concatenación de los textos de cada uno de los renglones de los objetos. En el caso de que el mensaje resultante exceda el tamaño del renglón, truncar el texto a la cantidad máxima de caracteres por renglón.

- Realizar la sobrecarga del operador << de modo tal que al utilizarlo en un objeto de tipo DisplayLCD, se imprima en la pantalla de nuestro embebido (no en el Display alfanumérico) el texto guardado en cada uno de los renglones con el siguiente formato:

  Renglón 1: TEXTO RENGLON 1
  Renglón 2: TEXTO RENGLON 2
  Renglón 3: TEXTO RENGLON 3
  Renglón 4: TEXTO RENGLON 4

Nota: Si lo desea puede utilizar como soporte las funciones de <string.h>