Problemes LP – Curs 2020-21
Tema 1: Programació orientada a objectes – Memòria dinàmica

Exercici 0.1 – nivell bàsic: Fet a classe.

Suposem que tenim creades les classes `Punt` I `Poligon` que hem utilitzat a la primera sessió de teoria, que serveixen per poder guardar un polígon a partir de la informació de cadascun dels seus vèrtexs:

```
class Punt
{
public:
    Punt();
    ~Punt();
    Punt(float x, float y): m_x(x), m_y(y) {};
    Punt(const Punt& pt);
    float getX() const { return m_x; }
    void setX(const float x) { m_x = x; }
    float getY() const { return m_y; }
    void setY(const float y) { m_y = y; }
private:
    float m_x;
    float m_y;
};
```

```
class Poligon
{
public:
    Poligon();
   ~Poligon();
   Poligon(int nCostats);
   int getNCostats() const { return m_nCostats; };
   bool afegeixVertex(const Punt &v);
   bool getVertex(int nVertex, Punt &v) const;
   float calculaPerimetre() const;
private:
    static const int MAX_COSTATS = 30;
    static const int MIN_COSTATS = 3;
    Punt m_vertexs[MAX_COSTATS+1];
    int m_nCostats;
    int m_nVertexs;
};
```

Volem modificar la declaració de la classe `Poligon` per utilitzar un array dinàmic per guardar la informació dels vèrtexs i així poder guardar polígons amb un nº qualsevol de vèrtexs, sense límit:

1. Feu els canvis necessaris als atributs de la classe `Poligon` per substituir l'array estàtic original per un array dinàmic.
2. Modifiqueu els dos constructors i el destructor de la classe Poligon per adaptar el codi a la utilització de l'array dinàmic. La resta de mètodes de la classe Poligon no s'haurien de modificar.
3. Afegiu un constructor de còpia i un operador d'assignació a la classe Poligon.

Tingueu en compte:

- A Caronte trobareu la declaració i implementació original de les classes `Punt` i `Poligon` a modificar. També trobareu un programa principal que serveix per provar i validar la implementació de les classes. Mireu-vos el codi per entendre'l i veure quins mètodes de la classe `Poligon` cal modificar quan afegiu l'array dinàmic.
- En la implementació original, els constructors suposen que un polígon té 3 un mínim de tres costats I un màxim de 30. En la nova implementació el mínim s'ha de mantenir, però el límit màxim ha de desaparèixer.

Exercici 0.2 (nivell mig): Fet a classe

Volem tornar a modificar la declaració de la classe `Poligon` de l'exercici **0.1** per guardar la informació dels vèrtexs utilitzant una estructura amb nodes dinàmics enllaçats:

1. Declareu i implementeu la classe `Node` que serà necessària per utilitzar una llista amb nodes dinàmics enllaçats.
2. Feu els canvis necessaris als atributs de la classe `Poligon` per substituir l'array dinàmic de l'exercici anterior per una estructura dinàmica amb nodes enllaçats.
3. Modifiqueu tots els mètodes de la classe `Poligon` (inclosos el constructor de còpia i l'operador d'assignació) per adaptar el codi a la utilització de l'estructura amb nodes dinàmics.

Tingueu en compte:

- El mètode `afegeixVertex` serveix per afegir un nou vèrtex al polígon. Suposem que els vèrtexs es van afegint seqüencialment des del primer fins a l'últim. Si ja s'ha afegit el màxim de vèrtexs que indica l'atribut m_nCostats, es retorna false, en cas contrari, true.
- El mètode `getVertex` serveix per recuperar la informació del vèrtex que ocupa la posició dins del polígon que es passa com a paràmetre a nVertex, on nVertex ha de tenir un valor entre 1 i el nº de vèrtexs ja afegits al polígon. En cas contrari es retorna fals.

Exercici 0.3 (nivell mig): Fet a classe

Volem tornar a modificar la declaració de la classe `Poligon` de l'exercici **0.2** per utilitzar un objecte de la classe de la llibreria estàndard `std::forward_list` per guardar la informació dels vèrtexs:

1. Feu els canvis necessaris als atributs de la classe `Poligon` per utilitzar la classe `std::forward_list` per guardar l'estructura dinàmica amb nodes enllaçats.
2. Modifiqueu tots els mètodes de la classe `Poligon` per adaptar el codi a la utilització dels mètodes de la classe `std::forward_list.`

Exercici 1 (nivell bàsic)

a) Implementeu una classe anomenada `Recta`, que permeti representar una recta i fer algunes operacions entre rectes i entre rectes i punts. Aquesta classe ha de tenir:

- Els atributs necessaris per guardar els paràmetres de la recta utilitzant l'equació general de la recta. Recordeu que l'equació general de la recta és de la forma $Ax + By + C = 0$. Per tant, necessitarem atributs per guardar els paràmetres A, B i C.
- Un constructor per defecte que inicialitzi els paràmetres a la recta amb equació $x = 0$.
- Un constructor per inicialitzar els paràmetres de la recta amb els valors que es passen com a paràmetre:

```
Recta(float a, float b, float c);
```

- Un constructor per inicialitzar els paràmetres de la recta perquè passi pels dos punts que es passen com a paràmetre:

```
Recta(const Punt& pt1, const Punt& pt2);
```

Suposant que (x_1, y_1) i (x_2, y_2) són els dos punts que es passen com a paràmetres, la recta (si no és vertical) queda definida per aquestes expressions:

$$A = \frac{y_2 - y_1}{x_2 - x_1}$$

$$B = -1.0$$

$$C = -Ax_1 + y_1$$

Si la recta és vertical $A = 1.0, B = 0$ i $C = -x_1$.

(*) Us donem ja feta la classe Punt que us permet guardar i recuperar les coordenades d'un punt.

- Mètodes per inicialitzar cadascun dels paràmetres de la recta:

```
void setA(float a);
void setB(float b);
      void setC(float c);
```

- Mètodes per recuperar els valors de cadascun dels paràmetres de la recta:

```
float getA() const;
float getB() const;
      float getC() const;
```

- Un mètode que transforma la representació de la recta a la seva forma canònica. La forma canònica de la recta és de la forma $Ax + By - 1 = 0$. Per tant, aquest mètode ha de re-escalar els paràmetres de la recta perquè C tingui el valor -1.

```
void formaCanonica();
```

- Un mètode per comprovar si dues rectes són paral·leles o no:

```
bool paralela(const Recta& r) const;
```

Recordeu que dues rectes seran paral·leles si els paràmetres A i B de les dues rectes guarden la mateixa proporció. Haureu de considerar també com a casos especials les rectes paral·leles a qualsevol dels dos eixos.

- La sobrecàrrega de l'operador de resta perquè calculi la distància entre la recta i un punt: per poder aplicar les operacions aritmètiques bàsiques: suma, resta, multiplicació i divisió.

  ```
  float operator-(const Punt &pt) const;
  ```

  Recordeu que la distància d'un punt a la recta es pot calcular amb l'expressió següent:

$$d = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}}$$

- La sobrecàrrega de l'operador de comparació que permeti determinar si dues rectes són iguals. Recordeu que dues rectes seran iguals si els tres paràmetres A, B i C de les dues rectes guarden la mateixa proporció. Tingueu en compte de tractar correctament els casos en què qualsevol dels 3 paràmetres val 0.

b) Utilitzeu la classe anterior per implementar una funció que permeti calcular totes les distàncies d'una recta a un conjunt de punts, amb la capçalera següent:

```
void distancia(Punt llistaPunts[], int nPunts, float distancies[]);
```

Primer de tot la funció ha de crear la recta que passa pels dos primers punts de l'array `llistaPunts` que es passa com a paràmetre. Després ha de calcular la distància de la resta de punts de l'array a la recta que s'acaba de crear i guardar totes les distàncies a l'array `distancies` que es passa com a paràmetre. `nPunts` és el nº de punts total de l'array llista Punts.

Exercici 2 (nivell bàsic): AVALUABLE

Volem crear una classe per poder guardar i manipular hores en format hores, minuts i segons. Aquesta classe haurà de tenir:

- Atributs per guardar l'hora, el minut i el segon.
- Un constructor per defecte.
- Un constructor que rebi com a paràmetres tres valors, l'hora, el minut i el segon i inicialitzi els atributs de la classe.
- Un constructor de còpia.
- Mètodes `setHora`, `setMinuts` i `setSegons`, per inicialitzar el valors del dia del mes i de l'any, respectivament
- Mètodes `getHora`, `getMinuts` i `getSegons` per retornar el valor del dia, del mes i de l'any, respectivament.
- Un mètode `horaValida` que comprovi si l'hora és una hora vàlida entre les 00:00:00 i les 23:59:59.
- Un operador + que permeti sumar un nombre de segons determinat a una data. Podeu suposar que el nombre de segons serà sempre positiu. Heu de tenir en compte que el nº de segons pot ser superior a un dia sencer.
- Un operador + que permeti sumar dues hores en format hora, minuts i segons (dos objectes de la classe `Hora`).
- Un operador == que permeti determinar si dues hores són iguals.
- Un operador < que permeti determinar si una hora és més petita que una altra.
- Un operador d'assignació.

Exercici 3 (nivell mig)

Volem implementar una versió simplificada del joc de la Oca. De moment, hem començat a crear una **classe `Jugador`** per guardar i gestionar la informació de cadascun dels jugadors que participen al joc i una **classe `Casella`** per poder gestionar les diferents caselles del tauler de joc i quines accions s'han de fer cada cop que un jugador cau a una casella.

La **classe `Jugador`** guarda tota la informació necessària de cada jugador pel desenvolupament del joc. Us donem feta la declaració de la classe i la implementació d'alguns mètodes bàsics:

```
class Jugador
{
public:
   Jugador() : m_casella(1), m_potTirar(true), m_nTornsInactiu(0),
         m_guanyador(false) {}
   int posicio() const { return m_casella; }
   bool getActiu() const { return m_potTirar; }
   void mou(int casella) { m_casella = casella; }
   void guanya() { m_guanyador = true; }
   bool esGuanyador() const { return m_guanyador; }
   void setInactiu(int nTorns);
   bool potTirar();
private:
   int m_casella;
   bool m_potTirar;
   int m_nTornsInactiu;
   bool m_guanyador;
};
```

Com podeu veure a nivell d'atributs, la classe té:

- Un atribut `m_casella` de tipus enter per guardar la posició del tauler on està en cada moment el jugador.
- Un atribut booleà `m_potTirar` que indica si el jugador pot tirar en el torn actual, o està atrapat en alguna casella que implica estar alguns torns sense tirar.
- Un atribut `m_nTornsInactius` que indica, si el jugador està atrapat en alguna casella que no li deixa tirar, quants torns han de passar per poder tornar a tirar.
- Un atribut booleà `m_guanyador` que només es posarà a `true` quan el jugador arribi a la casella final del joc.

A nivell de mètodes, apart del constructor, us donem fets aquests cinc mètodes:

- `posicio()`: retorna el número de casella on està el jugador.
- `getActiu()`: retorna si el jugador està actiu (pot tirar) o no.
- `mou(casella)`: aquest mètode es cridarà quan el jugador es mogui de casella. Serveix per canviar la posició del tauler on està el jugador.
- `guanya()`: indica que aquest és el jugador que guanya la partida.
- `esGuanyador()`: retorna si el jugador ha guanyat la partida o no.

Us demanem completar la definició de la classe `Jugador` implementant aquests dos mètodes:

- **`setInactiu(nTorns)`**: aquest mètode es cridarà cada cop que el jugador caigui en una casella que impliqui quedar-se uns quants torns sense poder tirar. Ha de canviar l'estat del jugador per indicar que en el següent torn no pot tirar i per guardar quants torns haurà d'estar el jugador sense poder tirar.
- **`potTirar()`**: aquest mètode ens retornarà si el jugador pot tirar en el torn actual, o no ho pot fer perquè està atrapat en alguna casella especial. En aquest cas, a més a més,

haurà de decrementar el número de torns que li queden per poder tornar a tirar. Quan el número de torns per tirar arribi a zero haurem de canviar l'estat per indicar que podem tornar a tirar.

La **classe `Casella`** guarda la informació bàsica de cada casella del joc i permet gestionar les accions que s'han de fer quan el jugador arriba a la casella, en funció del tipus de casella.

Us donem ja feta la declaració i la implementació d'alguns mètodes bàsics de la classe `Casella`.

```
const int NORMAL = 1;
const int OCA = 2;
const int POU = 3;
const int MORT = 4;
const int FINAL = 5;

class Casella
{
public:
    Casella() : m_posicio(0), m_tipus(NORMAL) {}
    void setPosicio(int posicio) { m_posicio = posicio; }
    void setTipus(int tipus) { m_tipus = tipus; }
    int getPosicio() const { return m_posicio; }
    bool esOca() const{ return (m_tipus == OCA); }
    bool entraJugador(Jugador& j, Casella caselles[], int nCaselles);
private:
    int m_posicio;
    int m_tipus;
};
```

Aquesta classe `Casella` té com a atributs `m_posicio`, que indica a quina posició del tauler correspon la casella i `m_tipus` que indica el tipus de la casella. En aquesta versió simplificada del joc només considerarem 5 tipus de caselles:

- **Casella normal**: quan el jugador arriba a la casella no passa res especial, simplement es canvia la posició del jugador a aquesta casella.
- **Oca**: quan el jugador arriba a la casella, salta fins a la següent casella marcada com a Oca i manté el torn.
- **Pou**: el jugador s'ha de quedar dos torns sense jugar.
- **Mort**: el jugador torna a la casella inicial.
- **Final**: casella final del joc. Si el jugador hi arriba guanya la partida.

Com a mètodes, us donem fets els getters i setters que poden fer falta.

Us demanem completar la definició de la classe `Casella`, **implementant el mètode `entraJugador`.** Aquest mètode es cridarà cada vegada que un jugador arriba a la casella i serveix per modificar l'estat del jugador que es passa com a paràmetre en funció de l'acció associada a la casella.

En el cas de les caselles normals, l'únic que s'ha de fer és cridar al mètode `mou` de la classe `Jugador` per modificar la posició a la que està el jugador.

En el cas de les caselles especials (oca, pou, mort, final), apart de moure el jugador a la posició de la casella, s'ha de modificar l'estat del jugador cridant als mètodes corresponents de la classe `Jugador` per reflectir l'acció associada a cada tipus de casella que hem explicat abans. En el cas de que la casella sigui una oca, el paràmetre `caselles` ha de servir per poder buscar la següent oca a la que hem de saltar.

Aquest mètode retornarà `true` si després de caure en aquesta casella el jugador conserva el torn (en aquesta versió simplificada només quan cau a una oca).

A partir de les definicions de les classes `Jugador` i `Casella` volem **implementar una classe `Tauler`** per poder guardar i gestionar tota la informació del joc.

Heu de declarar la **part privada** de la **classe `Tauler`** amb els atributs necessaris per poder guardar totes les caselles del joc (fins a un màxim de 63), les dades de tots els jugadors que hi participen (fins a un màxim de 4) i quin és el jugador que té el torn en cada moment del joc.

Heu d'implementar els mètodes següents de la **classe `Tauler`**:

- **Implementar** el **mètode `inicialitza`** de la **classe `Tauler`** amb la capçalera següent:

```
void inicialitza(const string& nomFitxer, int nJugadors);
```

  Aquest mètode ha d'inicialitzar les caselles del tauler llegint del fitxer que es passa com a paràmetre les dades de totes les caselles del joc. També ha de fixar i inicialitzar el nº de jugadors al valor que es passa com a paràmetre, i donat el torn actual al primer jugador. El fitxer amb les dades de les caselles tindrà aquest format:

```
POSICIO_1 TIPUS_1
POSICIO_2 TIPUS_1
...
POSICIO_N TIPUS_N
```

  POSICIO és un enter indicant el nº de la casella al tauler. Podeu suposar que les caselles estan ordenades al fitxer per la seva posició, des del 1 fins al nº total de caselles.
  TIPUS és un enter que indica el tipus de casella segons s'ha indicat a l'explicació de la classe `Casella`.

- **Implementar** el **mètode `tornJoc`** de la **classe `Tauler`** amb la capçalera següent:

```
void tornJoc(int valorDau);
```

  Aquest mètode ha de servir per actualitzar l'estat de la partida en funció del jugador que tingui el torn. Si el jugador que té el torn no pot tirar, no es fa res, simplement es salta el torn.

  Si el jugador actual pot tirar el paràmetre `valorDau` té valor del dau entre 1 i 6. S'ha de calcular la nova posició a la que ha d'anar el jugador i si la posició és més gran que la casella final tampoc es fa res, considerem que el jugador no es pot moure.

  Si es pot moure, s'haurà de moure el jugador a la casella corresponent i fer les accions associades al tipus de casella cridant al mètode `entraJugador` de la casella que heu implementat a l'exercici anterior. Com hem explicat a l'exercici 2 aquest mètode `entraJugador` retornarà `true` o `false` en funció de si el jugador conserva o no el torn. Si el jugador no conserva el torn s'ha de passar el torn al següent jugador modificant l'atribut corresponent de la classe `Tauler` que guarda el jugador que té el torn.

- **Implementar** el **mètode `getTipusCasella`** de la **classe `Tauler`** amb la capçalera següent:

```
int getTipusCasella(int nCasella);
```

  Aquest mètode ha de retornar el tipus de la casella de la posició que es passa com a paràmetre (començant per la posició 0).

- **Implementar** el **mètode `getEstatJugador`** de la **classe `Tauler`** amb la capçalera següent:

```
int getEstatJugador(int nJugador, int& posicio, bool& potTirar,
bool& guanyador);
```

Aquest mètode ha de retornar l'estat del jugador (posició del tauler, si pot tirar i si és el guanyador del joc) que es passa com a paràmetre (començant pel jugador 0) als paràmetres per referència `posició`, `potTirar` i `guanyador`.

Exercici 4 (nivell avançat): AVALUABLE

Volem implementar un conjunt de classes que ens permetin gestionar els lliuraments que els estudiants fan dels exercicis d'una assignatura en una plataforma similar a Caronte utilitzant l'eina de lliurament d'exercicis avaluables.

Per poder guardar la informació de cadascun dels exercicis que s'han de lliurar us donem ja creada una classe `Exercici` que guarda la descripció i la data límit de lliurament d'un exercici, un array dinàmic amb tots els lliuraments dels estudiants i el nº total d'estudiants que poden lliurar l'exercici:

```
class Exercici
{
public:
   Exercici();
   Exercici(const string& descripcio, const string& dataLimit);
   Exercici(const Exercici& e);
   ~Exercici();
   Exercici& operator=(const Exercici& e);
   void inicialitzaEstudiants(const string& fitxerEstudiants);
   bool afegeixTramesa(const string& niu, const string& fitxer,
         const string& data);
   bool consultaTramesa(const string& niu, const string& data,
         string& fitxer);
   bool eliminaTramesa(const string& niu, const string& data);
private:
   string m_descripcio;
   string m_dataLimit;
   LliuramentsEstudiant* m_lliuraments;
   int m_nEstudiants;
};
```

A l'array dinàmic `m_lliuraments` tindrem un element per cadascun dels estudiants que poden lliurar l'exercici. A cada element de l'array hi guardarem un objecte de la classe `LliuramentEstudiant` que guarda la informació de l'estudiant (niu i nota) i una llista amb totes les trameses que ha fet l'estudiant (un estudiant pot fer moltes trameses d'un mateix exercici, tantes com vulgui):

```
class LliuramentsEstudiant
{
public:
   LliuramentsEstudiant() : m_niu(""), m_nota(0.0) {};
   void setNiu(const string& niu) { m_niu = niu; }
   string getNiu() const { return m_niu; }
   float getNota() const { return m_nota; }
   void afegeixTramesa(const string& fitxer, const string& data);
   bool consultaTramesa(const string& data, string& fitxer);
   bool eliminaTramesa(const string& data);
private:
   string m_niu;
   float m_nota;
   forward_list<Tramesa> m_trameses;
};
```

Cada tramesa que fa l'estudiant consta d'un fitxer amb la solució de l'exercici i una data de lliurament:

```
class Tramesa
{
public:
   Tramesa();
   Tramesa(const string& fitxer, const string& data) : m_fitxer(fitxer),
m_data(data) {}
   void setFitxer(const string& fitxer) { m_fitxer = fitxer; }
   void setData(const string& data) { m_data = data; }
   string getFitxer() const { return m_fitxer; }
   string getData() const { return m_data; }
private:
   string m_fitxer;
   string m_data;
};
```

A partir d'aquestes definicions us demanem el següent:

1. **Implementeu els mètodes següents de la classe `LliuramentEstudiant`:**
   - **Implementeu el mètode `afegeixTramesa`** per afegir les dades d'una tramesa (nom del fitxer i data) a la llista de trameses de l'estudiant. El mètode tindrà aquesta capçalera:

     ```
     void afegeixTramesa(const string& fitxer, const string& data);
     ```

   - **Implementeu el mètode `eliminaTramesa`** per eliminar una tramesa que hagi fet l'estudiant. El mètode tindrà aquesta capçalera:

     ```
     bool eliminaTramesa(const string& data);
     ```

     S'ha d'eliminar la tramesa amb la data especificada com a paràmetre. Podeu suposar que un estudiant no farà dues trameses amb la mateixa data. El mètode retorna `true` si la tramesa existeix i es pot eliminar i `false`, en cas contrari.

   - **Implementeu el mètode `consultaTramesa`** per consultar una tramesa que hagi fet l'estudiant. El mètode tindrà aquesta capçalera:

     ```
     bool consultaTramesa(const string& data , string& fitxer);
     ```

     S'ha de retornar al paràmetre per referència `fitxer`, el nom del fitxer que correspon a la tramesa amb la data especificada com a paràmetre. Podeu suposar que un estudiant no farà dues trameses amb la mateixa data. El mètode retorna `true` si la tramesa existeix i `false`, en cas contrari.

2. **Implementeu els mètodes següents de la classe `Exercici`:**
   - **Implementeu el mètode `inicialitzaEstudiants`** per inicialitzar les dades dels estudiants que poden lliurar l'exercici a partir d'un fitxer de text. El mètode tindrà aquesta capçalera:

     ```
     void inicialitzaEstudiants(const string& fitxerEstudiants);
     ```

La primera línia del fitxer tindrà el nº total d'estudiants i a continuació hi haurà una línia per estudiant que contindrà només el niu de cadascun dels estudiants. S'haurà d'inicialitzar l'array dinàmic `m_lliuraments` amb el nº d'estudiants que indiqui la primera línia del fitxer i afegir a l'array un objecte del tipus `LliuramentsEstudiant` per cadascun dels estudiants del fitxer.

- **Implementeu el mètode `afegeixTramesa`** per afegir les dades d'una nova tramesa d'un estudiant. El mètode tindrà aquesta capçalera:

```
bool afegeixTramesa(const string& niu, const string& fitxer,
      const string& data);
```

S'ha de buscar l'estudiant amb el niu especificat dins de l'array `m_lliuraments` i, si existeix, afegir-hi una nova tramesa amb les dades (fitxer i data) que es passen com a paràmetre. Si l'estudiant existeix i es pot afegir la tramesa retorna `true`, i si no, retorna `false`.

Si és la primera tramesa que fa l'estudiant, s'haurà d'afegir un nou objecte de tipus `LliuramentEstudiant` al conjunt de lliuraments i afegir la tramesa corresponent als lliuraments d'aquest estudiant utilitzant el mètode `afegeixTramesa`. Si l'estudiant ja havia fet prèviament alguna tramesa, simplement s'haurà de recuperar l'objecte `LliuramentEstudiant` que correspon a aquest estudiant i afegir-hi la tramesa corresponent.

- **Implementeu el mètode `consultaTramesa`** per consultar les dades d'una tramesa d'un estudiant. El mètode tindrà aquesta capçalera:

```
bool consultaTramesa(const string& niu, const string& data,
      string& fitxer);
```

S'ha de buscar l'estudiant amb el niu especificat dins de l'array `m_lliuraments` i, si existeix, recuperar les dades (nom del fitxer) de la tramesa amb la data que es passa com a paràmetre. Si l'estudiant existeix i es pot recuperar la tramesa retorna `true`, i si no, retorna `false`.

- **Implementeu el mètode `eliminaTramesa`** per eliminar les dades d'una tramesa d'un estudiant. El mètode tindrà aquesta capçalera:

```
bool eliminaTramesa(const string& niu, const string& data);
```

S'ha de buscar l'estudiant amb el niu especificat dins de l'array `m_lliuraments` i, si existeix, eliminar les dades de la tramesa amb la data que es passa com a paràmetre. Si l'estudiant existeix i es pot eliminar la tramesa retorna `true`, i si no, retorna `false`.

- **Implementeu** el **constructor de còpia**, el **destructor** i l'**operador d'assignació**.

Exercici 5 (nivell mig)

Volem crear una classe `LlistaDoble` que implementi una llista doblement enllaçada similar a la classe `list`[1] de la llibreria estàndard. Recordeu que una llista doblement enllaçada és una estructura dinàmica amb nodes enllaçat on cada node de la llista té dos apuntadors, un a l'element següent i un altre a l'element anterior. Suposarem que els elements que es guarden a la llista són de tipus enter.

Volem que la classe `LlistaDoble` tingui un subconjunt dels mètodes que té la classe `list` de la llibreria estàndard. En concret, la classe ha de tenir aquesta interfície pública:

```
class LlistaDoble
{
    public:
        LlistaDoble();
        ~LlistaDoble();
        bool empty() const;
        Node* begin() const;
        Node* rbegin() const;
        Node* insert(const int &valor, Node* posicio);
        Node* erase(Node* posicio);
        void insertList(Node* posicio, int elements[], int nElements);
        void reverse();
        LlistaDoble& operator=(const LlistaDoble& llista);
 };
```

1. Implementeu la classe Node per guardar els elements de la llista, tenint en compte que ha d'incloure els dos apuntadors, a l'element següent i a l'element anterior.
2. Utilitzant la classe Node, afegiu els atributs necessaris a la classe `LlistaDoble` i implementeu tots els mètodes de la interfície pública de la classe que indiquem més amunt, tenint en compte que:
   - El mètodes `begin` i `rbegin` recuperen un apuntador al primer i a l'últim element de la llista, respectivament.
   - El mètode `empty` ha de retornar un booleà indicant si la llista està buida o no.
   - El mètode `insert` afegeix un element a la posició anterior del node passat com a paràmetre. Si l'apuntador que es passa com a paràmetre és `NULL` s'ha d'afegir al final de tot de la llilsta. Retorna un apuntador a l'element que s'ha afegit.
   - El mètode `erase` elimina l'element que ocupa la posició del node que es passa com a paràmetre. Si l'apuntador que es passa com a paràmetre és `NULL` no s'ha de fer res. Retorna un apuntador a l'element següent de l'element eliminat.
   - El mètode `insertList` ha d'afegir a la llista tots els elements de l'array que es passa com a paràmetre just a continuació del node `posicio` que també es passa com a parámetre. Si `posicio` té el valor `nullptr`, s'han d'afegir al principi de la llista.
   - El mètode `reverse` inverteix l'ordre dels elements de la llista.
   - L'operador d'assignació ha de copiar (assignant nova memòria) tots els elements de la llista que es passa com a paràmetre a la llista actual, alliberant primer tots els elements que hi hagués originalment.

[1] http://www.cplusplus.com/reference/list/list/

Exercici 6 (nivell avançat):AVALUABLE

Volem implementar una classe `Matriu` que permeti guardar una matriu de nombres reals de qualsevol dimensió i fer algunes operacions bàsiques amb els valors de la matriu. Aquesta classe ha de contenir:

- Tots els atributs necessaris per guardar una matriu dinàmica de nombres reals de qualsevol dimensió (la dimensió quedarà fixada quan s'inicialitzi la matriu).
- Un constructor per defecte que inicialitza una matriu buida amb dimensions (0,0).
- Un destructor.
- Un constructor amb dos paràmetres que siguin el nº de files i el nº de columnes de la matriu, que inicialitzi correctament la matriu a les dimensions indicades, creant tota la memòria dinàmica que sigui necessària. Tots els valors de la matriu han de quedar inicialitzats a 0.
- Un constructor de còpia que inicialitzi la matriu de forma que sigui exactament igual a la matriu que es passa com a paràmetre (tant en dimensions com en contingut).
- Un operador d'assignació que permeti assignar a la matriu el valor d'una altra matriu (`m1 = m2`) de forma que passin a ser iguals (tant en dimensions com en contingut). El contingut previ de la matriu es perd (i per tant, també s'ha d'alliberar la memòria dinàmica que tingués reservada).
- Un mètode `resize` que permeti modificar les dimensions de la matiu als valors que es passen com a paràmetre (creant la memòria dinàmica necessària). Igual que passa amb l'operador d'assignació, el contingut previ de la matriu (si n'hi havia) es perd (i per tant, també s'ha d'alliberar la memòria dinàmica que tingués reservada). Si les noves dimensions són més petites que les originals, els valors de la matriu han de quedar inicialitzats als mateixos valors que la matriu original per totes les posicions de la nova matriu que també existien a la matriu original. Si les dimensions són més grans (ja sigui per files o columnes), els nous valors s'han d'inicialitzar a 0. La capçalera d'aquest mètode ha de ser la següent:

  ```
  void resize(int nFiles, int nColumnes);
  ```

- Un mètode `transpose` que transposi la matriu (intercanviï files per columnes). S'ha d'alliberar la memòria dinàmica de la matriu original i assignar nova memòria dinàmica per les noves dimensions intercanviades de la matriu. La capçalera d'aquest mètode ha de ser la següent:

  ```
  void transpose(int nFiles, int nColumnes);
  ```

- Un mètode `setValor` per modificar el valor de la matriu en una posició determinada. Si la fila i columna que s'indiquen com a posició a modificar són més grans que el tamany de la matriu no s'ha de fer res.
- Un mètode `getValor` per modificar el valor de la matriu en una posició determinada. Si la fila i columna que s'indiquen com a posició a recuperar són més grans que el tamany de la matriu s'ha de retornar el valor especial NAN.
- Mètodes `getNFiles` i `getNColumnes` per recuperar el nº de files i de columnes de la matriu, respectivament.

- Un mètode `esBuida` que retorni si la matriu està buida (encara no s'ha iniciallitzat a unes dimensions determinades).
- La sobrecàrrega de l'operador + de suma perquè rebi com a paràmetre una matriu i retorni el resultat de sumar la matriu actual amb la matriu que es passa com a paràmetre. Si les dimensions de les dues matrius no són iguals s'ha de retornar una matriu buida.
- Una segona sobrecàrrega de l'operador + de suma perquè rebi com a paràmetre un nº real i retorni el resultat de sumar aquest valor a tots els valors de la matriu.

Exercici 7 (nivell avançat)

Volem fer un programa que simuli el sistema de preinscripció universitària i l'assignació de places a les diferents titulacions de les universitats catalanes. Per això, necessitarem primer guardar les dades de tots els estudiants que volen inscriure's a la universitat. Cada estudiant haurà de proporcionar les seves dades personals (DNI i nom), la nota de selectivitat que ha obtingut i un llistat amb els estudis que voldria cursar. Aquest llistat contindrà vuit opcions (nom del títol i nom de la universitat on el voldria fer) ordenades per ordre de preferència.

A més a més, necessitarem també les dades de totes les universitats i titulacions del sistema universitari català. Per cada universitat guardarem el nom i el nº de titulacions que ofereix. Per cada titulació, el nom, el nº de places que ofereix, el nº de places assignades i també un llistat amb el DNI de tots els estudiants assignats.

Suposarem que tenim un fitxer amb la informació de tots els estudiants (DNI, nom i nota de selectivitat, ordenats segons la nota de selectivitat en ordre descendent, i les vuit opcions escollides) i un altre fitxer amb la informació de totes les universitats i titulacions (nom universitat i nº de titulacions i nom i nº de places ofertes per cadascuna de les titulacions). Aquests fitxers serviran per inicialitzar les dades dels estudiants i les titulacions al principi del programa. El nombre total d'estudiants no el podem saber a priori. En canvi, sí que sabem que en tot el sistema universitari públic català hi ha un total de 8 universitats.

Amb aquestes dades ja podem iniciar el procés d'assignació de places. Aquest procés funcionarà de la forma següent: començarem assignant plaça a l'estudiant que hagi tingut la nota de selectivitat més alta i continuarem per la resta d'estudiants segons la nota en ordre descendent. Per cada estudiant, començarem per la primera opció que ha escollit. Si en aquesta opció (títol i universitat) encara queden places lliures, li assignarem la plaça, l'afegirem com a pre-admès en aquests estudis i decrementarem el nombre de places lliures. Si no queden places lliures, mirarem les opcions restants per ordre de preferència fins que en trobem alguna on hi hagi places lliures. Si no n'hi ha cap l'alumne es quedarà sense plaça.

El programa ha de començar llegint les dades de tots els estudiants i les titulacions dels fitxers que contenen aquesta informació. Després ha de realitzar el procés d'assignació de places tal com s'ha descrit prèviament. Al final s'escriurà en un fitxer anomenat "assignacio.txt" el resultat de l'assignació. En aquest fitxer, per cada titulació s'escriurà primer una línia amb el nom de la universitat, el nom de la titulació i el nº d'estudiants assignats a la titulació, i a continuació un llistat amb el DNI de tots els estudiants assignats, un estudiant per línia del fitxer, ordenats per nota de selectivitat en ordre descendent. En el fitxer les titulacions han d'aparèixer seguint el mateix ordre que tenien en el fitxer inicial des d'on es llegeixen les dades.

Heu de crear totes les classes necessàries amb els atributs i mètodes que calgui per poder fer el procés d'assignació tal com s'ha descrit abans. A Caronte teniu el programa principal de prova que llegeix estudiants i titulacions, fa l'assignació i escriu el resultat a fitxer. Aquest programa suposa que existeix una classe anomenada `SistemaUniversitari` amb els mètodes necessaris per llegir estudiants, llegir titulacions, fer l'assignació i escriure el resultat. Fixeu-vos en els noms que han de tenir aquests mètodes. Aquesta classe l'haureu d'omplir amb els atributs necessaris per guardar tota la informació per fer l'assignació.

A Caronte trobareu també exemples dels fitxers inicials d'estudiants i titulacions. Fixeu-vos en el format dels fitxers per implementar les funcions que llegeixen les dades. També trobareu un exemple del format del fitxer de sortida "assignació.txt". Assegureu-vos que la sortida del vostre programa segueix exactament aquest format.

Si ho necessiteu, podeu utilitzar les classes `forward_list` i `list` de la llibreria estàndard.