```cpp
//1r - En la clase parte publica
friend ostream& operator<<(ostream& out, Matriu& m);
//2n - hacer su definicion correspondiente
//segundo parametro el objeto que vamos a hacer la sobrecarga
ostream& operator<<(ostream& out, Matriu& m)
{
    for (int i = 0; i < m.m_vecX.size(); i++)//en out add las cadenas
        out << "add texto que queramos mostrar";
    return out;//finalmente devolvemos el conjunto de cadenas
}
// ---------------- Lectura ficheros ----------------------
#include<fstream>
void leerFichero(nombreFichero)
{
    ifstream miFichero; //Declaro el objeto ifstream para poder leer el fichero
    miFichero.open(nomFitxer);//abrir ficheros

    if (miFichero.is_open())
    {
        int fila, col;
        string cadena;
        while (!miFichero.eof()) // mientras no sea el final del fichero
        {
            ...
            //se guarda con el tipo de variable de llegada
            miFichero >> fila >> col >> cadena;
            ...
        }
        miFichero.close();
    }
}
```

```cpp
bool LliuramentsEstudiant::eliminaTramesa(const string& data)
{
    bool eliminarTramesa = false;
    std::forward_list<Tramesa>::iterator itAnterior = m_trameses.before_begin();
    std::forward_list<Tramesa>::iterator itActual = m_trameses.begin();
    while (!eliminarTramesa && (itActual != m_trameses.end()))
    {
        if (itActual->getData() == data)
        {
            eliminarTramesa = true;
            itActual = m_trameses.erase_after(itAnterior);
        }
        else
        {
            itAnterior = itActual;
            itActual++;
        }
    }
    return eliminarTramesa;
}
```

```cpp
template<class T>
inline T* SmartPointer<T>::operator->()
{
    cout << "op->" << endl;
    if (pointer == NULL)
        cout << "ERROR; APUNTADOR A NULL" << endl;

    return pointer;
}
```

El procés d'herència es pot fer de dues maneres:

- public

```cpp
class Llibre: public Producte
```

- private

```cpp
class Llibre: private Producte
```

	Herència **public**		Herència **private**	
Qualificació a la classe base	**Accés des de classes no derivades**	**Accés des de classe derivada**	**Qualificació a la classe derivada**	**Accés des de classe derivada**	**Qualificació a la classe derivada**
private	no accessible	no accessible	private	no accessible	private
protected	no accessible	accessible	protected	accessible	private
public	accessible	accessible	public	accessible	private

Constructors:
1. Crida al constructor per defecte de la classe base
2. Crida al constructor especific de la classe derivada

Destructors:
1. Crida al destructor de la classe derivada
2. Crida al destructor de la classe base

```cpp
void Llibre::mostra()
{
    Producte::mostra();
    cout << "Titol: " << m
    cout << "Autor: " << m
    cout << "N. pagines: " << m_nPagines << endl;
}
```

- Crida explícita a mostra de la classe base
- S'ha de posar si volem mostrar informació de la classe base

```cpp
class Llibre: public P
{
public:
    ...
    void mostra();
    ...
```

- Mètode de la classe derivada
- Mostra informació específica de la classe derivada
- Substitueix la definició de mostra de la classe base

Un objecte de la classe derivada es pot convertir a un objecte de la classe base encara que es perdi informació

```
Objecte_classe_base = Objecte_classe_derivada
```

Un objecte de la classe base NO es pot convertir a un objecte de la classe derivada perquè no disposem de valor per tots els atributs específics

```
Objecte_classe_derivada = Objecte_classe_base
```

- Una classe derivada no **té** accés a les dades privades de la classe base
- Si ens interessa accedir a la part privada de la classe base:
  - Declarar els atributs/mètodes a la classe base com a **protected**:
    - Les classes derivades hi poden accedir com si fossin `public`
    - La resta de classes no derivades ho segueixen veient com a `private` i per tant, no hi poden accedir

```cpp
class Producte
{
    ...
    virtual void mostra();
    ...
};
```

Declaració del mètode com a virtual

```cpp
class Llibre: public Producte
{
    ...
    void mostra();
    ...
};
```

```cpp
class Electrodomestic : public Producte
{
    ...
    void mostra();
    ...
};
```

Sense utilitzar apuntadors

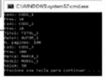
```cpp
list<Producte> llistaProductes;
llistaProductes.push_back(p);
llistaProductes.push_back(l);
llistaProductes.push_back(e);
list<Producte>::iterator it;
for (it = llistaProductes.begin();
  it != llistaProductes.end(); it++)
  it->mostra();
```

Utilitzant apuntadors

```cpp
list<Producte*> llistaProductes;
llistaProductes.push_back(&p);
llistaProductes.push_back(&l);
llistaProductes.push_back(&e);
list<Producte*>::iterator it;
for (it = llistaProductes.begin();
  it != llistaProductes.end(); it++)
  (*it)->mostra();
```

Crida sempre a mètode mostra de la classe Producte

Crida al mètode mostra de la clase utilitzada per inicialitzar l'apuntador



Destructors virtuals

```cpp
bool LlistaProductes::eliminaProducte(co
{
  ...
  if ((*it)->getCodi() == codi)
      delete *it;
      m_productes.erase(it);
      return true;
  else
      it++;
}
```

```cpp
class Producte
{
  ...
  virtual ~Producte();
}
```

```
Eliminant Llibre
Destructor Llibre
Destructor Producte
Eliminant Electrodomestic
Destructor Electrodomestic
Destructor Producte
```

```cpp
cout << "Eliminant Llibre" << endl;
llista.eliminaProducte(l1.getCodi());
cout << "Eliminant Electrodomestic" << endl;
llista.eliminaProducte(el.getCodi());
```

- Al declarar el destructor com a virtual dinàmicament es crida al destructor de la classe derivada de l'objecte corresponent (`Llibre` / `Electrodomestic`)
- Recordem que primer es crida al destructor de la classe derivada (`Llibre` / `Electrodomestic`) i després al de la classe base (`Producte`)
- Els constructors no es poden declarar com a virtuals (sempre es crida al constructor de l'objecte que estem creant)

```cpp
class Producte
{
public:
    ...
    virtual Producte* clone() { return new Producte(*this); }
    ...
}
```

- A la classe base afegim un mètode virtual `clone()`
- Retorna un apuntador a un nou objecte dinàmic de la classe inicialitzat com una còpia de l'objecte actual

```cpp
class Llibre: public Producte
{
public:
    ...
    Llibre* clone() { return new Llibre(*this); }
    ...
}
```

- A la classe derivada redefinim aquest mètode canviant el tipus de retorn de l'apuntador i canviant el tipus de l'objecte dinàmic que es crea

```cpp
class Producte
{
public:
    ...
    virtual Producte* clone() = 0;
    virtual float calculaPreu(int nUnitats) = 0;
    virtual float calculaDespesesEnviament() = 0;
    virtual void mostra() = 0;
private:
    ...
};
```

```cpp
void Producte::mostra()
{
    cout << "Codi: " << m_codi << endl;
    cout << "Preu: " << m_preu << endl;
}
```

```cpp
void Llibre::mostra()
{
    Producte::mostra();
    ...
}
```

```cpp
virtual funcio() = 0;
```

Funcions virtuals pures

- Mètodes que no s'utilitzen a la classe base
- Mètodes que volem que la classe derivada redefineixi obligatòriament
- Podem definir codi pel mètode de la classe base (p.ex. `calculaDespesesEnviament`, `mostra`), però no cal si no és necessari (p.ex. `clone`, `calculaPreu`)
- Les classes derivades han de redefinir el codi del mètode. Si no ho fan el mètode també passa a ser una funció virtual pura a la classe derivada

```cpp
template<class TClau, class TValor>
class Map
{
public:
    Map();
    Map(const Map<TClau, TValor>& m);
    ~Map() {}

    int longitud() const;
    bool esBuit() const;
    Map<TClau, TValor>& operator=(const Map< TClau, TValor>& m);

    TValor& operator[](const TClau& clau);
    const TValor& operator[](const TClau& clau) const;
    TClau& operator[](const int& posicio);
    const TClau& operator[](const int& posicio) const;

    void afegeix(const TClau& clau, const TValor& valor);
private:
    vector<pair<TClau, TValor>> m_vector;
    TClau defectoC;
    TValor defectoV;
};
```

```cpp
template<class T>
inline T& SmartPointer<T>::operator*()
{
    cout << "op*" << endl;

    if (pointer == NULL)
        cout << "ERROR; APUNTADOR A NULL"<<endl;

    return *pointer;
}
```

```cpp
template<class T>
```

```cpp
template<class TClau, class TValor>
inline Map<TClau, TValor>& Map<TClau, TValor>::operator=(const Map<TClau, TValor>& m)
{
    if (this != &m)
    {
        int newsize = m.m_vector.size();
        m_vector.resize(newsize);

        for (int i = 0; i < newsize; i++)
            m_vector[i] = m.m_vector[i];
    }

    return *this;
}

template<class TClau, class TValor>
inline TValor & Map<TClau, TValor>::operator[](const TClau & clau)
{
    int t = m_vector.size() - 1;
    int b = 0;
    int m;

    bool trobat = false;

    while (b <= t && !trobat)
    {
        m = (b + t);
        m /= 2;
        if (clau == m_vector[m].first)
        {
            trobat = true;
            return m_vector[m].second;
        }
        else
        {
            if (clau < m_vector[m].first)
                t = m - 1;
            else
                b = m + 1;
        }
    }

    if (!trobat)
        return defectoV;
}
```

```cpp
template<class TClau, class TValor>
ne const TClau & Map<TClau, TValor>::operator[](const int & posicio) const

    int size = m_vector.size();

    if ((posicio > 0) && (posicio < size))
        return m_vector[posicio].first;
    else
        return defectoC;
```

```cpp
template<class TClau, class TValor>
inline void Map<TClau, TValor>::afegeix(const TClau & clau, const TValor & valor)
{
    bool trobat = false;

    typename vector<pair<TClau, TValor>>::iterator actual = m_vector.begin();

    while (actual != m_vector.end() && !trobat)
    {
        if (clau < actual->first)
        {
            trobat = true;
            pair<TClau, TValor> nuevo;
            nuevo = make_pair(clau, valor);
            m_vector.insert(actual, nuevo);
        }
        else
        {
            if (clau == actual->first)
            {
                trobat = true;
                actual->second = valor;
            }
            else
                actual++;
        }
    }
}
```