PUBLICACIÓ DOCENT

MANUAL DE LABORATORI D'ESIN Sessió 1

AUTOR: Bernardino Casas, Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Q3
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
ANY: 2019

Vilanova i la Geltrú, 30 d'agost de 2019

1

Exercici

L'objectiu d'aquest exercici és implementar la classe `mcj_enters` que ens permeti representar i manipular multiconjunts (finits) d'enters usant memòria estàtica. Un multiconjunt és un conjunt on els elements poden estar repetits (i com succeeix en els conjunts, no importa l'ordre dels elements). De fet, la classe només ens permetrà representar multiconjunts de com a màxim `MAXSIZE` elements. El fitxer `mcj_enters.hpp` amb l'especificació de la classe es descriu amb detall en la següent subsecció.

Observa com es comporten les operacions d'unió, intersecció i diferència de multiconjunts amb aquest exemple:

A:	{2 2 3 1 1}
B:	{4 1 2 2 2}
A ∩ B:	{2 2 1}
A ∪ B:	{4 2 2 2 3 1 1}
A - B:	{3 1}
B - A:	{4 2}

Bàsicament el que cal fer és:

1. trobar una representació adequada pels objectes de la classe i escriure els atributs necessaris en la part `private` de `mcj_enters.hpp`;

2. començar amb una implementació trivial per tots els mètodes en `mcj_enters.cpp`, i després implementar i provar els diferents mètodes paulatinament. Els primers mètodes que hauries d'implementar són la constructora, `insereix`, `conte` i `print`.

En cas que no es tingui clar el passos a seguir per crear un classe en `C++` es recomana llegir el Decàleg (veure l'apèndix B).

1.1 Especificació

Aquesta és l'especificació de la classe `mcj_enters` que cal copiar en el fitxer `mcj_enters.hpp` (te'l pots copiar de la carpeta `/home/public/esin/sessio1`):

```
#ifndef _MCJ_ENTERS_HPP_
#define _MCJ_ENTERS_HPP_
#include <iostream>
using namespace std;

// Un objecte de la classe mcj_enters representa a un multiconjunt
// de com a màxim MAXSIZE enters.
class mcj_enters {
public:
  // Constructora per defecte. Crea un multiconjunt buit.
  mcj_enters();

  // Les tres grans: Constructora per còpia, destructora, operador d'assignació
  mcj_enters(const mcj_enters &cj);
  ~mcj_enters();
  mcj_enters& operator=(const mcj_enters &cj);

  // Insereix l'enter e en el multiconjunt.
  void insereix(int e);

  // Unió intersecció diferència de multiconjunts. Operen modificant el multiconjunt
  // sobre el que s'aplica el mètode, usant el segon multiconjunt com argument.
  // Per ex.: a.restar(b) fa que el nou valor d'a sigui A - B, on A i B són
  // els valors originals dels objectes a i b.
  void unir(const mcj_enters& B);
  void intersectar(const mcj_enters& B);
  void restar(const mcj_enters& B);

  // Unió, intersecció i diferència de multiconjunts. Operen creant un nou
  // multiconjunt sense modificar el multiconjunt sobre el que s'aplica el mètode.
  // La suma de multiconjunts correspon a la unió, la resta a la diferència i
  // el producte a la intersecció.
  mcj_enters operator+(const mcj_enters& B) const;
  mcj_enters operator*(const mcj_enters& B) const;
  mcj_enters operator-(const mcj_enters& B) const;

  // Retorna cert si i només si e pertany al multiconjunt.
  bool conte(int e) const;

  // Retornen els elements màxim i mínim del multiconjunt, respectivament.
  // El seu comportament no està definit si el multiconjunt és buit.
  int max() const;
  int min() const;

  // Retorna el nombre d'elements (la cardinalitat) del multiconjunt.
```

```
46    int card() const;
47
48    // Operadors relacionals. == retorna cert si i només si els dos multiconjunts
49    // (el paràmetre implícit i B) contenen els mateixos elements;
50    // != retorna cert si i només si els multiconjunts són diferents.
51    bool operator==(const mcj_enters& B) const;
52    bool operator!=(const mcj_enters& B) const;
53
54    // Imprimeix el multiconjunt d'enters, ordenats en ordre ascendent, sobre
55    // el canal de sortida os; el format és [e1 e2 ... en], és a dir, amb
56    // espais entre els elements i tancant la seqüència amb corxets.
57    void print(ostream& os) const;
58
59  private:
60    // Aquí van els atributs i mètodes privats
61  };
```

Si treballes amb portàtil o amb el PC de casa teva, pots accedir remotament al servidor `ahto` de l'escola on tens el mateix entorn que en els PCs de les aules informàtiques. Ho pots fer executant la següent comanda en una terminal, on usuari és el teu usuari de linux (habitualment el teu DNI canviant el primer número per una lletra):

```
ssh usuari@ahto.epsevg.upc.edu
```

Recorda que a la carpeta `/home/public/esin` hi ha tot el material de l'assignatura ESIN.

Si prefereixes copiar tots els fitxers d'ESIN del servidor `ahto` en el teu portàtil o PC, ho pots fer executant la següent comanda en el teu PC:

```
scp -r usuari@ahto.epsevg.upc.edu:/home/public/esin/* .
```

1.2 Implementació i testeig

Un cop completada la part privada de la classe `mcj_enters` en el fitxer `mcj_enters.hpp` i implementats els mètodes públics i privats en el fitxer `mcj_enters.cpp` caldria fer un programa principal en el fitxer `main.cpp` que crei alguns objectes de la classe `mcj_enters`, insereixi dades i testegi els diferents mètodes aplicant-los sobre aquests objectes. Després compilar, linkar i testejar.

Per facilitar la feina de testeig, dins del curs ESIN (Vilanova) de jutge.org us hem creat el problema "Classe multiconjunt d'enters" (https://jutge.org/problems/X39772_ca) on només cal enviar l'especificació i implementació de la classe `mcj_enters`. Degut a que jutge.org només permet l'enviament d'un fitxer amb la solució del problema, en el mateix fitxer hi ha d'haver l'especificació i la implementació de la classe (el que normalment estarien separats en els fitxers .hpp i .cpp). Els pots ajuntar amb la comanda:

```
cat mcj_enters.hpp mcj_enters.cpp > solucio.cpp
```

i enviar a jutge.org el fitxer `solucio.cpp`. Prèviament, en el fitxer `mcj_enters.cpp` cal eliminar la directiva `#include "mcj_enters.hpp"` per no tenir problemes de precompilació.

Aquest problema "Classe multiconjunt d'enters" ja disposa d'un programa principal i de jocs de prova públics i privats que automatitza la feina de testeig. Un cop enviïs la teva solució, jutge.org farà la compilació i linkat de tot i testejarà que passin tots els jocs de prova.

Recorda que implementarem el multiconjunt d'enters usant memòria estàtica (tots els elements del multiconjunt d'enters estaran guardats en posicions de memòria consecutives, típicament en un vector). Et pots inspirar en les implementacions de piles i cues amb un vector que trobaràs en el "Tema 3. Estructures lineals estàtiques" de teoria.

Com que aquesta versió de `mcj_enters` amb memòria estàtica té un límit en el màxim nombre d'element d'un multiconjunt, segurament no passarà els jocs de prova privats on hi ha casos amb multiconjunts molt grans (a no ser que `MAXSIZE` el defineixis amb un valor d'almenys 20000 elements).

Si els mètodes d'unir, intersectar, restar, igualtat i diferència no es programen de forma eficient (cost lineal) tampoc passaran els jocs de prova privats degut a un excés en el temps d'execució. Per tant has de calcular el cost temporal que té cadascun dels mètodes que has implementat i, en cas que sigui pitjor que un cost lineal, caldrà pensar com millorar la implementació, potser retocant com es guarden els elements del multiconjunt d'enters en la memòria estàtica.

Envia la solució a jutge.org amb l'anotació "Fet amb memòria estàtica" perquè el professor sàpiga quina versió mirar quan te la corregeixi.

A

Punters, pas de paràmetres i taules

A.1 Punters

En temps d'execució, la declaració d'una variable de cert tipus implica la reserva d'un espai de memòria per valors d'aquest tipus. Aquesta reserva de memòria comença en una direcció de memòria concreta.

Tenint en compte això:

- L'operador `&` sobre una variable retorna la direcció on comença l'espai reservat per aquesta variable.
- La direcció d'una variable és del tipus associat `T*`. A una variable del tipus `T*` se l'anomena tipus punter a `T` i pot contenir direccions de variables del tipus `T`.
- L'operador `*` aplicat a un punter permet accedir a la informació d'aquest punter.

Per veure un exemple d'ús de punters veure la figura A.1.

```
1 int x, z;
2 int *y;
3
4 x = 12;
5 y = &x;   // p.e. la direcció de x pot ser 6240fa102
6 z = *y;   // z valdrà 12 (el mateix que x)
7
8 *y = 1480; // x valdrà 1480
9 y = x+z;    // ara y tindrà una direcció incorrecta: 1492
```

Un punter que no apunta a cap lloc s'anomena *punter nul* i el seu valor és 0 (≡`NULL`). Si `p` és un punter nul llavors `*p` no està definit, i en general provoca un error d'execució immediat (tipus *Segmentation fault*), però tot dependrà del sistema. En qualsevol cas és un error greu.

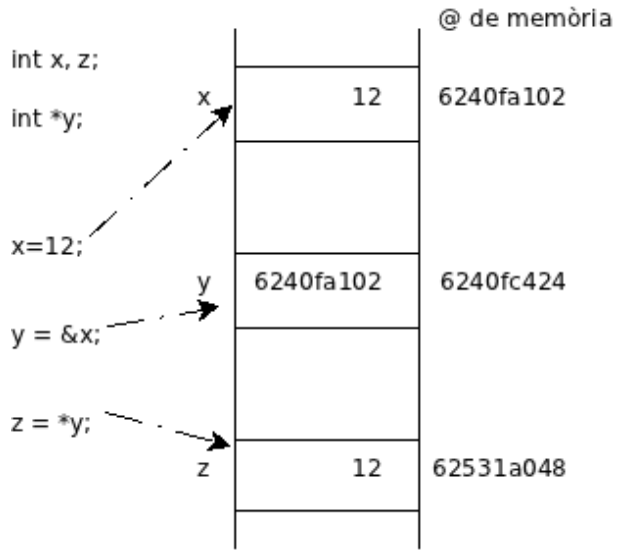


Figura A.1: Exemple gràfic de l'ús de punters.

```
1 char *p;
2 char c = 'b';
3 *p = 'a'; // ERROR! p no està inicialitzat, pot apuntar a qualsevol lloc!
4 p = 0;     // p és nul
5 p = &c;    // p apunta a c
6 *p = 0;    // c conté un caràcter amb codi ASCII 0
```

Es poden declarar punters genèrics del tipus `void*` . Aquests punters poden apuntar a qualsevol cosa:

```
1 int x;
2 float y;
3 void *g = &x;
4 ...
5 g = &y;
```

Convé evitar el seu ús, ja que són una font d'errors molt difícil de corregir. El seu ús ha de quedar restringit a funcions de molt baix nivell.

A.2 Referències

Una referència és un nom alternatiu a una variable ja declarada, és a dir, un alias. Una referència d'una variable de tipus `T` és de tipus `T&`.

Tota referència cal que sigui inicialitzada en la seva declaració excepte quan la referència és un paràmetre d'una funció. En aquest darrer cas la referència s'inicialitza automàticament quan s'invoca la funció i s'efectua el pas de paràmetres.

```
1 char a;
2 char& b = a;
```

El valor d'una referència no es pot canviar. Només es pot canviar el valor a la que es refereix la referència.

```
1 a = 'Z';
2 b = 'X';    // a ara val 'X'
```

La direcció d'una referència és la mateixa de la variable a la qual es refereix.
Les referències estan implementades mitjançant punters, però no disposen de les operacions sobre punters.

A.3 Pas de paràmetres

Els paràmetres de sortida i d'entrada/sortida s'especifiquen mitjançant l'ús de referències, és a dir, amb el *pas de paràmetres per referència* on:

- Paràmetre formal: `T& identificador`
- Paràmetre actual: `variable`

Per altra banda, els paràmetres d'entrada es poden especificar de tres maneres diferents:

1. *Pas de paràmetre per valor*: el paràmetre formal es comporta com una variable local de la funció, inicialitzada amb el valor del paràmetre actual (s'usa el constructor per còpia del tipus T):
   - Paràmetre formal: `T identificador`
   - Paràmetre actual: `expressió`
2. *Pas de paràmetre per valor constant*: el paràmetre formal es comporta com una constant local de la funció, inicialitzada amb el valor del paràmetre actual (s'usa el constructor per còpia del tipus T):
   - Paràmetre formal: `const T identificador`
   - Paràmetre actual: `expressió`
3. *Pas de paràmetre per referència constant*: el paràmetre formal és una referència al paràmetre actual (NO es fa cap còpia) però el paràmetre formal no es pot modificar:
   - Paràmetre formal: `const T& identificador`
   - Paràmetre actual: `variable`

Es recomana el **pas de paràmetres per valor** (`T x`, `const T x`) pels paràmetres elementals predefinits (`int`, `char`, `bool`, ...). En canvi, pels paràmetres d'entrada que siguin objectes "compostos" és preferible el **pas de paràmetres per referència constant** (`const T& x`) per així evitar fer còpies costoses.

El qualificatiu `const` permet que el compilador detecti errors en els que es modifica inadvertidament un paràmetre d'entrada. Això no seria un problema si usem pas per valor (`T x`), ja que treballem amb una còpia, però és un problema molt seriós si usem pas per referència, doncs la funció treballa amb el paràmetre actual (simplement se li dóna un altre nom o alias per referir-se a ell).

Els següents exemples permeten apronfundir en el mecanisme de pas de paràmetres:

Exemple 1) Què s'escriurà per pantalla quan s'executi el següent codi?

```
1  void proc(int x) {
2      cout << x << endl;
3      x = 14;
4      cout << x << endl;
5  }
6
7  int main() {
8      int a = 10;
9      cout << a << endl;
10     proc (a);
11     cout << a << endl;   // a no ha canviat el seu valor
12 }
```

Exemple 2) I si afegim **&** al paràmetre de l'acció *proc*? Què passaria?

```
1  void proc(int &x) {
2      cout << x << endl;
3      x = 14;
4      cout << x << endl;
5  }
6
7  int main() {
8      int a = 10;
9      cout << a << endl;
10     proc (a);
11     cout << a << endl;   // a val 14
12 }
```

Exemple 3) I si la capçalera de l'acció *proc* fos aquesta:

```
void proc (const int &x) {
    ...
}
```

Què passaria?

```
void proc(const int &x) {
    cout << x << endl;
    x = 14;
    cout << x << endl;
}

int main() {
    int a = 10;
    cout << a << endl;
    proc (a);
    cout << a << endl;   // ERROR de compilació
}
```

Els paràmetres de sortida o d'entrada/sortida poden ser simulats mitjançant l'ús de punters a l'estil de C. Fer-ho d'aquesta manera no és recomanable.

A.4 Retorn de resultats

Una funció pot retornar els seus resultats per còpia, per referència o per referència constant:

tipus funcio(llista_params_formals)

on *tipus* és el nom d'un tipus (`T` o `void`), una referència (`T&`), o una referència constant (`const T&`).

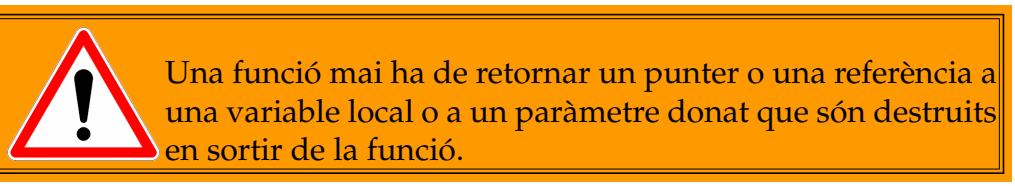
```
int f1(int x) {
    return x;
}

int f2(int& x) {
    return x;
}

int& f3(int& x) {
    return x;
}

int& f4(int x) {   // MALAMENT!!
```

```
14     return x;
15 }
16
17 void f() {
18     int w = 10;
19
20     int y = f1(w);
21     // Amb aquesta crida es generen 3 còpies del valor w:
22     // w → x, x → resultat, resultat → y
23
24     int z = f2(w);
25     // El paràmetre formal necessàriament ha de ser una variable.
26     // Es generen 2 còpies del valor w:
27     // w = x → resultat, resultat → z
28
29     int u = f3(w);
30     // genera 1 còpia: w = x = resultat → u
31 }
```



El compilador detecta la major part d'errors com els que mostrem a continuació si activem els flags adequats. Però, a vegades no són detectats i els errors es produeixen en temps d'execució.

```
1  float* calcula(int x, const float y) {
2      float z = 0.0;
3      while (x > 0) {
4          z = z + y / x;
5          --x;
6      }
7      return &z;      // ERROR!!
8  }
9
10 float& calcula(int x, const float y) {
11     float z = 0.0;
12     while (x > 0) {
13         z = z + y / x;
14         --x;
15     }
16     return z;      // ERROR!!
17 }
```

A.5 Taules (arrays)

C++ té un constructor de vectors (arrays) predefinit que bàsicament funciona igual que C o Java.

Els arrays (i els punters) solen utilitzar-se com mecanismes de baix nivell per implementar classes. Aquests detalls d'implementació queden amagats a l'usuari, que utilitzarà classes segures (es realitzen les comprovacions que siguin necessàries internament) i no haurà de manipular arrays directament.

A.5.1 Declaració

Per declarar un vector de *n* elements del tipus T escriurem:

T identificador[n];

El valor *n* ha de ser constant o una expressió calculable en temps de compilació. Per crear un *array dinàmic* s'ha d'utilitzar una tècnica diferent (veure la sessió 2).

Els components d'un vector d'*n* elements s'indexen de 0 fins a *n* - 1.

En C++ no es fa cap comprovació de rang ni estàtica ni en temps d'execució. Donat un array A amb *n* elements, accedir a A[i], si i ¡ 0 ó i ¿= *n* pot provocar un error immediat o tenir conseqüències encara pitjors.

A.5.2 Consulta/Modificació de les taules

Per consultar una de les caselles d'un vector cal indicar el nom del vector i entre cortxets el número de la casella.

identificador[index];

L'*índex* està comprès entre 0 . . . numero_elements - 1.
Exemple:

```
v[0]
v[1]
...
v[19]
```

o també:

```
v[0] = 15;
```

A.5.3 Connexió entre arrays i punters

La connexió entre arrays i punters en C++ es profunda: un array és de fet un punter constant al primer component del vector: p ≡ & p[0]. En general, p + i ≡ & p[i].

L'única diferència entre un punter i un array és que un array no pot ser "modificat":

```
int p[10];
int z[10];
int* q = p;
for (int i = 0; i < 10; ++i) {
    p[i] = (i + 1) * 10;
}
cout << *q << endl;          // imprimeix 10
cout << p[2] << endl;        // imprimeix 30
cout << *(q + 2) << endl;    // imprimeix 30
q[2] = 33;
cout << *(p + 2) << endl;    // imprimeix 33
q = z;   // és correcte
p = z;   // ERROR! la direcció guardada a p no pot ser modificada
```

Donat que un array és un punter, es pot passar com paràmetre eficientment, però caldrà usar el qualificador `const` per evitar que es facin modificacions accidentals si es tracta d'un paràmetre d'entrada.

Un array només es pot passar per referència o per referència constant (mai es pot passar un array per valor), amb les mateixes regles de pas de paràmetres que en la resta de casos.

El tipus d'un array de `T`'s és, sigui quina sigui la mida, `T* const` o de manera equivalent `T[]`.

No existeixen els arrays multidimensionals. Per crear matrius caldrà usar un array d'arrays.

```
double mat[10][20];
```

B

Decàleg per implementar una classe en C++

Començar a implementar una classe en C++ des de cero pot ser un procés complicat els primers cops que ho feu. En aquest capítol veurem els passos que caldria seguir per tal d'implementar una classe en C++. Per tal d'il·lustrar el procés amb més detall es desenvoluparà un exemple: la classe `IntCell`. El concepte d'aquesta classe és molt senzill, ja que només ens permet emmagatzemar un únic enter dins la classe.

1ª Llei Crear la capçalera de la classe

FER el fitxer de capçalera de la classe (a partir d'ara l'anomenarem **fitxer .HPP**) on apareixen els mètodes (constructors, modificadors i consultors) que permeten actuar sobre una classe.

En el cas de les pràctiques d'ESIN ja disposeu del fitxer capçalera en el Campus Digital[1]. Per tant no l'heu de crear. És a dir, en aquest punt heu de tenir el fitxer capçalera de la classe similar al que es pot veure a la figura B.1.

2ª Llei Pensar la representació

PENSAR la representació interna de la classe.

- ⋆ La majoria de classes han d'emmagatzemar algun tipus d'informació. O sigui, s'ha de pensar ***com guardar aquesta informació***.
- ⋆ Les possibles representacions d'una classe van des de les més simples (variable, struct, ...) fins a representacions més complicades (taules, arbres, ...).
- ⋆ A l'hora de triar la representació d'una classe s'ha de pensar en l'eficiència esperada de cada una de les operacions.

[1] http://atenea.upc.edu

```
#ifndef _INTCELL_HPP_
#define _INTCELL_HPP_

#include <string>

class IntCell {
  public:
    explicit IntCell(int initialValue=0);

    int read() const;
    void write(int x);
    IntCell operator+(const IntCell &ic) const;

  private:
    // Si la classe fos per una sessió de laboratori la representació s'afegiria aquí.

  };
  #endif
```

Figura B.1: Exemple: Fitxer *intcell.hpp*

⋆ Una vegada s'ha decidit quina és la representació adient, és bo escriure-la sobre paper, i així acabar de visualitzar completament les estructures que s'hagin pensat. En el cas de representacions complexes és aconsellable (a més d'escriure l'explicació) fer una *representació gràfica*.

3ª Llei Escriure la representació

ESCRIURE la representació de la classe un cop ja tenim les idees ***clares***. La representació interna d'una classe normalment s'escriurà dins el fitxer `.HPP` (a la part privada). A la pràctica de cara a poder corregir les classes de manera automàtica, la declararem en un fitxer apart que anomenarem **fitxer .REP**. És a dir:

- *Sessions de Laboratori*: la representació de la classe es posarà en el fitxer `.HPP`.
- *Pràctica*: s'ha de crear un fitxer amb el mateix nom de la classe però amb extensió `.REP`. Aquest fitxer contindrà les variables, structs, tipus, ..., que representen la nostra classe. Veure la figura B.2.

És convenient documentar acuradament els fitxers per tal de deixar clar QUÈ s'ha fet i PER QUÈ s'ha fet d'aquesta manera.

```
[...]
private:
   // La representació de la classe IntCell està formada per un únic atribut
   // "_storedValue", on emmagatzemem el valor de la cel·la.
   int _storedValue;
};
#endif
```

Figura B.2: Exemple: Fitxer *intcell.rep*

4ª Llei Crear fitxer .CPP

- COPIAR el **fitxer .HPP** de la classe com a **fitxer .CPP** (fitxer d'implementació). Aquesta comanda a Linux és la següent:

  ```
  cp fitxer.hpp fitxer.cpp
  ```

- Tot seguit, hem d'ESBORRAR del fitxer `.CPP` **TOT** el contingut **menys les capçaleres de les operacions públiques** que hem d'implementar. Observeu la figura B.3 per tenir un exemple del resultat esperat d'aquest pas després d'haver transformat el fitxer capçalera.

```
explicit IntCell(int initialValue=0);
int read() const;
void write(int x);
IntCell operator+(const IntCell &ic) const;
```

Figura B.3: Resultat d'aplicar aplicar la *4a llei* sobre *intcell.hpp*

5ª Llei Retocar les capçaleres de les operacions

1. INCLOURE **davant del nom de cadascuna de les operacions** el nom de la classe seguit de l'operador scope (**::**). Veure la figura B.4.

```
explicit IntCell::IntCell(int initialValue=0);
int IntCell::read() const;
void IntCell::write(int x);
IntCell IntCell::operator+(const IntCell &ic) const;
```

Figura B.4: Fitxer .CPP *després* del primer pas de la *5ª llei*

2. AFEGIR al final del nom del mètode { } i esborrar el **;**. Veure la figura B.5.

```
1 explicit IntCell::IntCell(int initialValue=0) { }
2 int IntCell::read() const { }
3 void IntCell::write(int x) { }
4 IntCell IntCell::operator+(const IntCell &ic) const { }
```

Figura B.5: Fitxer .CPP *després* del segon pas de la *5ª llei*

3. ESBORRAR les inicialitzacions per defecte[2] `(=0, =1, ...)` i el modificador `explicit`[3] de les capçaleres del `.CPP`. Veure la figura B.6.

```
1 IntCell::IntCell(int initialValue) { }
2 int IntCell::read() const { }
3 void IntCell::write(int x) { }
```

Figura B.6: Fitxer .CPP *després* del tercer pas de la *5ª llei*

6ª Llei Afegir l'include

INCLOURE al principi del fitxer `.CPP` un include amb el nom del fitxer `.HPP` de la classe que estem creant:

```
#include "nom-classe.hpp"
```

El resultat d'aplicar aquesta llei a l'exemple es pot veure en la figura B.7.

[2] Les inicialitzacions per defecte o també dites paràmetres per defecte especifiquen quin valor ha de prendre un paràmetre en cas que no s'indiqui el seu valor. Els paràmetres per defecte només figuraran en l'especificació de la classe, és a dir, el fitxer `.HPP`.

[3] El modificador `explicit` indica que no es poden aplicar conversions de tipus implícites. Per exemple, si la constructora de la classe `IntCell` estigués declarada amb el modificador `explicit` en aquest cas:

```
IntCell ic;
ic = 37;
```

el compilador donaria un error ja que els tipus no coincideixen. En cas contrari el compilador no es queixaria i l'assignació mitjançant la conversió s'hauria produït. Aquest modificador només pot aparèixer al fitxer `.HPP`.

```
#include "intcell.hpp"

IntCell::IntCell(int initialValue) { }
int IntCell::read() const { }
void IntCell::write(int x) { }
IntCell IntCell::operator+(const IntCell &ic) const { }
```

Figura B.7: Fitxer .CPP *després* de la *6ª llei*

7ª Llei Primera compilació

- En aquests moments el fitxer .CPP ha de COMPILAR sense cap error.

  ```
  g++ -c -Wall nom-classe.cpp
  ```

  Sols cal compilar la classe i no cal linkar-la amb un programa principal ja que no farà res. Per tenir més detalls de com es compila/linka en C++ mirar l'**apèndix B** d'aquest manual.

- Quan compilem pot ser que apareguin més d'un *warning* ja que els mètodes de la classe no fan res, i alguns mètodes poden necessitar retornar quelcom.

- Si apareixen *errors* vol dir que hem realitzat algun dels passos malament. Revisar tots els passos.

8ª Llei Implementació incremental

IMPLEMENTAR una operació de la classe. És aconsellable començar per la/les constructora/es per continuar amb les modificadores i acabar amb les consultores. En qualsevol cas és aconsellable implementar una única operació alhora.

9ª Llei Compilar, Linkar i Provar

Sempre després d'acabar d'implementar un mètode cal compilar, i sempre que es pugui provar el seu funcioment. Si ho fem així podrem estalviar-nos que en una compilació ens sortin 300 errors, o que un error de funcionament de la classe estigui a l'operació constructora (al principi de tot).

Per comprovar una classe cal provar-la generalment amb un programa que tingui un **main**. Es pot veure un exemple de programa principal en la figura B.8.

S'ha de compilar el programa principal, i muntar-ho tot (linkar) per generar l'executable.

```
g++ -o nom_executable.e nom-classe1.o nom-classe2.o
```

```cpp
#include <iostream>
#include "intcell.hpp"

using namespace std;

int main () {
  IntCell icell;
  cout << icell.read() << endl;
}
```

Figura B.8: Fitxer *prog.cpp* per provar la constructora per defecte de la classe IntCell

Un cop s'ha comprovat que l'operació funciona correctament cal tornar a aplicar la **8ª llei** amb una altra operació.

10ª Llei Proves globals

Un cop acabada la implementació de totes les operacions de la classe hem de PROVAR-la amb els casos normals i els límit. Es pot veure la implementació acabada de la classe que estem fent servir com a exemple en la figura B.9.

```cpp
#include "intcell.hpp"

IntCell::IntCell(int initialValue) {
   _storedValue = initialValue;
}

int IntCell::read() const {
   return _storedValue;
}

void IntCell::write(int x) {
   _storedValue = x;
}

IntCell IntCell::operator+(const IntCell &ic) const {
   IntCell nou(_storedValue + ic._storeredValue);
   return nou;
}
```

Figura B.9: Fitxer *intcell.cpp* acabat