



**UNIVERSITAT POLITÈCNICA DE CATALUNYA**  
**BARCELONATECH**

**Escola Politècnica Superior d'Enginyeria  
de Vilanova i la Geltrú**

**PUBLICACIÓ DOCENT**

# **MANUAL DE LABORATORI D'ESIN**

**AUTOR:** Bernardino Casas

**ASSIGNATURA:** Estructura de la Informació (ESIN)

**CURS:** Segon

**TITULACIONS:** Grau en Informàtica

**DEPARTAMENT:** Ciències de la Computació

**ANY:** 2017/2018

Vilanova i la Geltrú, 18 de setembre de 2017



# Índex curt

<b>0</b>	<b>Punters, pas de paràmetres i taules</b>	<b>1</b>
<b>1</b>	<b>Memòria dinàmica</b>	<b>9</b>
<b>2</b>	<b>Excepcions i genericitat</b>	<b>27</b>
<b>3</b>	<b>La biblioteca estàndard de C++</b>	<b>47</b>
<b>A</b>	<b>Decàleg per implementar una classe en C++</b>	<b>69</b>
<b>B</b>	<b>Compilació, muntatge i execució en C++</b>	<b>75</b>
<b>C</b>	<b>Estil de programació i documentació</b>	<b>83</b>
	<b>Índex alfabètic</b>	<b>99</b>



# Índex

<b>0</b>	<b>Punters, pas de paràmetres i taules</b>	<b>1</b>
0.1	Punters . . . . .	1
0.2	Referències . . . . .	2
0.3	Pas de paràmetres . . . . .	3
0.4	Retorn de resultats . . . . .	5
0.5	Taules (arrays) . . . . .	7
0.5.1	Declaració . . . . .	7
0.5.2	Consulta/Modificació de les taules . . . . .	7
0.5.3	Connexió entre arrays i punters . . . . .	8
<b>1</b>	<b>Memòria dinàmica</b>	<b>9</b>
1.1	Introducció . . . . .	9
1.2	Reserva i alliberament de memòria dinàmica . . . . .	9
1.3	Problemes amb la gestió de la memòria dinàmica . . . . .	11
1.4	Constructor per còpia . . . . .	14
1.5	Destructor . . . . .	15
1.6	Operador d'assignació . . . . .	16
1.7	El component this . . . . .	17
1.8	Exemple pila . . . . .	18
1.8.1	Especificació de la classe pila . . . . .	18
1.8.2	Implementació de la classe pila . . . . .	18
1.8.3	Programa d'exemple que usa la classe pila . . . . .	20
1.9	Exercici . . . . .	21
1.9.1	cj_enters.hpp . . . . .	21
1.9.2	Activitats . . . . .	23
<b>2</b>	<b>Excepcions i genericitat</b>	<b>27</b>
2.1	Gestió d'errors . . . . .	27
2.1.1	Avortar el programa . . . . .	28
2.1.2	Codis d'error . . . . .	28
2.1.3	Assercions . . . . .	29
2.2	Excepcions en C++ . . . . .	29
2.2.1	Detecció i generació d'una excepció . . . . .	29
2.2.2	Tractament de les excepcions . . . . .	30
2.2.3	Propagar una excepció . . . . .	32
2.3	Classe error . . . . .	33

2.4	Genericitat . . . . .	35
2.4.1	Plantilles de funcions . . . . .	35
2.4.2	Plantilles de classes . . . . .	37
2.5	Exercici . . . . .	40
2.5.1	conjunt.hpp . . . . .	41
2.5.2	Activitats . . . . .	43
<b>3</b>	<b>La biblioteca estàndard de C++</b>	<b>47</b>
3.1	Introducció a l'STL . . . . .	47
3.2	La classe string . . . . .	48
3.2.1	Ús bàsic . . . . .	48
3.2.2	Operadors + i += . . . . .	49
3.2.3	Longitud . . . . .	49
3.2.4	Operadors de comparació . . . . .	49
3.2.5	Mètode substr . . . . .	50
3.2.6	Mètode replace . . . . .	50
3.2.7	Mètode find . . . . .	50
3.3	vector<T> . . . . .	51
3.3.1	Ús bàsic . . . . .	52
3.3.2	Iteradors . . . . .	52
3.3.3	Capacitat . . . . .	53
3.3.4	Accés als elements . . . . .	54
3.3.5	Modificadors . . . . .	55
3.4	list<T> . . . . .	56
3.4.1	Ús bàsic . . . . .	56
3.4.2	Iteradors . . . . .	57
3.4.3	Capacitat . . . . .	58
3.4.4	Accés als elements . . . . .	59
3.4.5	Modificadors . . . . .	59
3.4.6	Altres operacions . . . . .	60
3.5	Exercici . . . . .	61
3.5.1	Algorisme de Huffman . . . . .	62
3.5.2	arbbin.hpp . . . . .	62
3.5.3	huffman.hpp . . . . .	66
<b>A</b>	<b>Decàleg per implementar una classe en C++</b>	<b>69</b>
1 <sup>a</sup>	Llei Crear la capçalera de la classe . . . . .	69
2 <sup>a</sup>	Llei Pensar la representació . . . . .	69
3 <sup>a</sup>	Llei Escriure la representació . . . . .	70
4 <sup>a</sup>	Llei Crear fitxer .CPP . . . . .	71
5 <sup>a</sup>	Llei Retocar les capçaleres de les operacions . . . . .	71
6 <sup>a</sup>	Llei Afegir l'include . . . . .	72
7 <sup>a</sup>	Llei Primera compilació . . . . .	73
8 <sup>a</sup>	Llei Implementació incremental . . . . .	73
9 <sup>a</sup>	Llei Compilar, Linkar i Provar . . . . .	73
10 <sup>a</sup>	Llei Proves globals . . . . .	74

<b>B</b>	<b>Compilació, muntatge i execució en C++</b>	<b>75</b>
B.1	Compilació separada i muntatge . . . . .	75
B.1.1	Compilació i muntatge bàsic . . . . .	75
B.1.2	Compilació i muntatge de classes genèriques . . . . .	76
B.1.3	Compilació i muntatge amb biblioteques . . . . .	79
B.1.4	Altres opcions . . . . .	79
B.2	Make . . . . .	80
B.3	Execució . . . . .	81
<b>C</b>	<b>Estil de programació i documentació</b>	<b>83</b>
C.1	Noms de variables adequats . . . . .	83
C.2	Conveni consistent pels identificadors . . . . .	84
C.3	Utilitzar noms en forma activa per les funcions . . . . .	84
C.4	Nom d'identificadors precisos . . . . .	86
C.5	Identació del codi . . . . .	86
C.6	Evitar una lògica del programa antinatural . . . . .	89
C.7	Disminuir la complexitat . . . . .	91
C.8	Useu construccions similars per tasques similars . . . . .	92
C.9	No usar variables globals . . . . .	93
C.10	Utilitzar variables locals . . . . .	95
C.11	Codi ben estructurat . . . . .	96
C.12	Bona documentació . . . . .	97
	<b>Índex alfabètic</b>	<b>99</b>





# Índex de figures

1	Exemple gràfic de l'ús de punters. . . . .	2
3.1	Procés per crear l'arbre de codificació de Huffman. . . . .	63
A.1	Exemple: Fitxer <i>intcell.hpp</i> . . . . .	70
A.2	Exemple: Fitxer <i>intcell.rep</i> . . . . .	71
A.3	Resultat d'aplicar aplicar la 4a llei sobre <i>intcell.hpp</i> . . . . .	71
A.4	Fitxer <i>.CPP</i> després del primer pas de la 5a llei . . . . .	71
A.5	Fitxer <i>.CPP</i> després del segon pas de la 5a llei . . . . .	72
A.6	Fitxer <i>.CPP</i> després del tercer pas de la 5a llei . . . . .	72
A.7	Fitxer <i>.CPP</i> després de la 6a llei . . . . .	72
A.8	Fitxer <i>prog.cpp</i> per provar la constructora per defecte de la classe <i>IntCell</i> .	73
A.9	Fitxer <i>intcell.cpp</i> acabat . . . . .	74
B.1	Compilació i muntatge de les classes <i>a</i> i <i>b</i> , i el programa principal <i>prog.</i> . .	77
B.2	Compilació i muntatge de les classes <i>a</i> (genèrica) i <i>b</i> , i el programa principal <i>prog.</i> . . . .	78



# 0

## Punters, pas de paràmetres i taules

### 0.1 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 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 ( $\equiv$ 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.

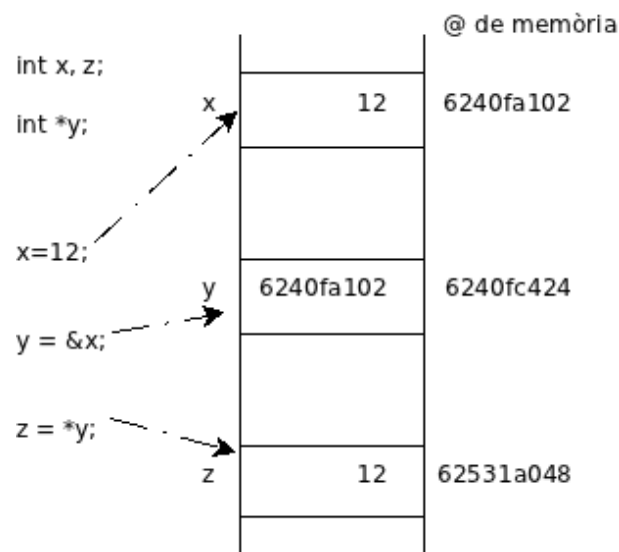


Figura 1: Exemple gràfic de l'ús de punters.

```

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.

## 0.2 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.

## 0.3 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:

```

1 void proc (const int &x) {
2     ...
3 }

```

Què passaria?

```

1 void proc(const 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;  // ERROR de compilació
12 }

```

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.

## 0.4 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&).

```

1 int f1(int x) {
2     return x;
3 }
4
5 int f2(int& x) {
6     return x;
7 }
8
9 int& f3(int& x) {
10    return x;
11 }
12
13 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 }

```



Una funció mai ha de retornar un punter o una referència a una variable local o a un paràmetre donat que són destruïts en sortir de la funció.

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 }

```



## 0.5 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.

### 0.5.1 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  $i \geq n$  pot provocar un error immediat o tenir conseqüències encara pitjors.

### 0.5.2 Consulta/Modificació de les taules

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

```
identificador[index];
```

L'*index* està comprès entre  $0 \dots \text{numero\_elements} - 1$ .

*Exemple:*

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

o també:

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

### 0.5.3 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:  $p \equiv \&p[0]$ . En general,  $p + i \equiv \&p[i]$ .

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

```

1 int p[10];
2 int z[10];
3 int* q = p;
4 for (int i = 0; i < 10; ++i) {
5     p[i] = (i + 1) * 10;
6 }
7 cout << *q << endl;           // imprimeix 10
8 cout << p[2] << endl;         // imprimeix 30
9 cout << *(q + 2) << endl;     // imprimeix 30
10 q[2] = 33;
11 cout << *(p + 2) << endl;    // imprimeix 33
12 q = z; // és correcte
13 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];
```

# 1

## Memòria dinàmica

### 1.1 Introducció

La memòria dinàmica permet la reserva i l'alliberament d'espai de memòria per dades durant l'execució del programa, és a dir, ens permet un ús eficient de l'espai de memòria i utilitzar només la quantitat necessària. L'ús de memòria dinàmica pot portar a simplificar el codi en alguns problemes i a complicar el mateix en d'altres. Per això, existeix un compromís entre l'ús de memòria estàtica i dinàmica.

### 1.2 Reserva i alliberament de memòria dinàmica

Per crear i destruir objectes en memòria dinàmica s'utilitzen els operadors:

- ★ `new` / `delete`: reserva o allibera una porció de memòria.
- ★ `new[]` / `delete[]`: reserva o allibera un array de  $n$  objectes (taules dinàmiques).

Si un punter  $q$  apunta a una taula creada amb `new[]`, la memòria ha de ser alliberada amb `delete[] q`. Anàlogament, si  $p$  apunta a un objecte creat amb `new`, llavors l'objecte s'ha de destruir amb `delete p`. És important tenir en compte que els objectes creats amb memòria dinàmica són "anònims", és a dir, accessibles únicament a través de punters.

Un objecte creat amb memòria dinàmica existeix fins que no sigui destruït explícitament (o acabi l'execució del programa).

En el següent codi es pot veure un exemple d'obtenció i alliberament de memòria dinàmica per tipus predefinits:

```
1 int* pi = new int;  
2 // reserva memòria per un enter al qual apunta pi  
3  
4 char *pc = new char[10];  
5 // reserva memòria per una taula de 10 caràcters [0..9]
```

```

6 // pc apunta a la component 0 (pc ≡ &pc[0])
7 // pc[i] ≡ *(pc+i) accedeix a la i-èsima component
8
9 ...
10
11 delete pi;
12 // allibera la memòria de l'enter apuntat per pi; el punter conserva el seu
13 // valor, però ara apunta a una zona de memòria disponible per reservar.
14
15 delete[] pc;
16 // allibera la memòria de la taula de caràcters apuntada per pc

```

Els operadors `new` y `delete` no es limiten a crear l'espai de memòria o alliberar-lo. Una cop creat l'espai necessari per l'objecte, `new` invoca al constructor adequat per inicialitzar al nou objecte. Quan es crea una taula d'objectes amb `new[]` el sistema invoca automàticament el constructor per cadascun d'ells.

Per la seva banda, `delete` aplica el destructor de la classe a la que pertany l'objecte apuntat i després allibera la memòria ocupada per l'objecte. Quan es destrueix una taula d'objectes el sistema invoca el destructor per cadascun d'ells en ordre invers a la seva creació.

```

1
2 class vector {
3     public:
4         vector(int s = 8);           // vector de s enters
5         vector(const vector &v);    // constructor per còpia
6         ~vector();                  // destructor
7         ...
8 };
9
10 int main() {
11     vector* pv1 = new vector(12);
12     // crea un vector de 12 enters apuntat per pv1
13
14     vector* pv2 = new vector;
15     // crea un vector de 8 enters apuntat per pv2
16
17     vector* ptv1 = new vector[10];
18     // crea una taula de 10 vectors de 8 enters apuntada per ptv1
19
20     vector* pv3 = new vector(*pv2);
21     // crea un còpia de *pv2 apuntat per pv3
22
23     vector* ptv2 = new vector[10](*pv1);
24     // crea una taula de 10 vectors on cada un dels vectors és una còpia de
25     // *pv1 i està apuntada per ptv2

```

```

26
27     delete pv1;
28     delete pv2;
29     delete[] ptv1;
30     delete pv3;
31     delete[] ptv2;
32     // es destrueix la memòria assignada per tots els punters
33 }

```

## 1.3 Problemes amb la gestió de la memòria dinàmica

Alguns problemes comuns en la gestió de la memòria dinàmica inclouen:

- **Emparellament incorrecte:** No “aparellar” correctament els operadors.  
Per exemple: intentar destruir amb `delete` una taula creada amb `new[]`.
- **Dangling references:** Alliberar un objecte que no ha estat creat amb memòria dinàmica o ja ha estat alliberat.  
Per exemple:

```

1 int x;
2 int* p = new int;
3 int* q = &x;
4 delete q; // ERROR: q no apunta a un objecte creat usant memòria dinàmica
5 q = p;
6 delete p; // OK
7 delete q; // ERROR: l'objecte al que apuntava q ha deixat d'existir

```

- **Memory leaks:** Perdre l'accés a objectes creats amb memòria dinàmica i per tant no tenir la possibilitat de destruir-los.  
Per exemple:

```

1 void f() {
2     int* p = new int;
3     *p = 3;
4 }
5 // ERROR! quan finalitza l'execució de la funció f la variable local p deixa
6 // d'existir i no tenim forma d'accedir a l'objecte.
7
8 int f2() {
9     int* p = new int;
10    int* q = new int;
11    p = q; // ERROR! perdem l'accés al primer objecte
12    ...
13 }
14

```

```

15 int* g(int x) {
16     int* p = new int;
17     *p = x;
18     return p;
19 }
20 // OK: es pot "recollir" el punter a l'objecte creat dinàmicament en el punt de
21 // crida a la funció g; però aquesta forma de treballar no és aconsellable.
22
23 void h(node* p) {
24     node* q = new node;
25     p -> seg = q;
26 }
27 // OK: es té accés al nou node a partir del punter seg que és apuntat per un
28 // punter extern a la funció h (el paràmetre de la funció).

```

- **Desreferència de NULL:** Dereferenciar (amb \* o amb ->) un punter nul. Aquest error té quasi sempre resultats catastròfics (*segmentation fault*, *bus error*, ...). Per exemple:

```

1 bool esta(const llista& l, int x) {
2     node* p = l.primer;
3     while (p != NULL and p -> info != x) {
4         p = p -> sig;
5     }
6     return (p -> info == x);
7     // ERROR! si p == NULL, p->info és incorrecte!
8 }

```

- **Problema de l'aliasing:** Usar o implementar incorrectament constructores per còpia o l'operador d'assignació per classes implementades mitjançant memòria dinàmica. Per exemple:

```

1 char s[] = "abc"; // s[0] = 'a', s[1] = 'b', s[2] = 'c',
2                // s[3] = '\0'
3 char t[10];
4 t = s;
5 cout << t[0]; // imprimeix 'a'
6 t[0] = 'b';
7 cout << s[0]; // imprimeix 'b'!! t és en realitat un char* i l'assignació
8                // t = s fa que t apunti a 's[0]'.

```

O també:

```

1 class estudiant {
2     public:
3         estudiant(char* nom, int dni);
4         char* consulta_nom();
5         int consulta_dni();

```

```

6     ...
7     private:
8         char* _nom;
9         int _dni;
10 };
11
12 int main() {
13     estudiante a("pepe", 45218922);
14
15     estudiante b = a; // el constructor per còpia d'ofici no serveix!
16
17     a = b; // l'operador d'assignació d'ofici és inadequat!
18 }

```

- **Retornar punter local:** Retornar un punter o una referència a un objecte local d'una funció. El problema és que a l'acabar la funció, l'objecte local es destrueix i el punter o referència deixa d'apuntar a un objecte vàlid. Per exemple:

```

1 int& maxim(int A[], int n) {
2     int max = 0;
3     for (int i = 0; i < n; ++i) {
4         if (A[i] >= A[max]) {
5             max = i;
6         }
7     }
8     return A[max]; // OK: es retorna una referència a A[max]
9 }
10
11 int& maxim(int A[], int n) {
12     int max = 0;
13     for (int i = 0; i < n; ++i) {
14         if (A[i] >= max) {
15             max = A[i];
16         }
17     }
18     return max; // ERROR: es retorna una referència a una variable local!
19 }

```

- **Delete de NULL:** Fer delete  $p$  amb  $p == NULL$  no és erroni. No té cap efecte i ocasionalment el seu ús ajuda a simplificar el codi.

### REGLA DELS TRES GRANS

La regla del tres grans (*anglès: the Law of the Big Three*) indica que si necessites una implementació no trivial del:

- constructor per còpia,
- destructor, o
- operador d'assignació.

segurament necessitaràs implementar els altres. Aquests tres mètodes són automàticament creats pel compilador si no són explícitament declarats pel programador. SEMPRES que una classe emperi memòria dinàmica caldrà implementar aquests tres mètodes, ja que les implementacions d'ofici amb punters no funcionen com nosaltres voldríem. Així doncs, per ser més flexibles, en les nostres especificacions posarem aquests tres mètodes.

## 1.4 Constructor per còpia

Un constructor per còpia inicialitza objectes amb còpies d'altres objectes de la mateixa classe. Aquest mètode té el mateix nom que la classe, rep com a paràmetre un objecte de la mateixa classe i com els mètodes constructor no retorna res. El perfil d'aquest mètode per una classe *X* seria:

```
X::X(const X&) {
    ...
}
```

Tota classe té un constructor per còpia. Si no el programem, el compilador proporciona un "d'ofici" que es limita a copiar un a un els atributs. En general, el constructor per còpia "d'ofici" ens va bé, exceptuant si l'objecte té un o més atributs que són punters a memòria dinàmica o un array.

Cal tenir en compte que el constructor per còpia s'invoca automàticament quan fem:

- pas de paràmetres per valor
- retorns per valor
- declaracions del tipus:

```
vector v1 = v2;
vector v1(v2);
```



Exemple de constructor per còpia:

```

1 class vector {
2     public:
3         vector(int s = 8);
4         vector(const vector& v); // per còpia
5         ...
6     private:
7         int *_t; // punter de la taula d'enters
8         int _s; // mida de la taula
9 };
10
11 vector::vector(int s) : _s(s), _t(new int[s]) {}
12
13 vector::vector(vector& v) : _s(v._s), _t(new int[v._s]) {
14     for (int i=0; i < _s; ++i) {
15         _t[i] = v._t[i];
16     }
17 }
18
19 void f() {
20     vector v1(10); // vector de 10 enters
21     ...
22     vector v2 = v1; // v2 és còpia de v1
23 }

```

## 1.5 Destructor

Els destructors proporcionen un mecanisme automàtic que garantitza la destrucció dels objectes. Es declaren com mètodes que no retornen cap resultat i el nom del mètode destructor és el nom de la classe precedit del caràcter ~. Els mètodes destructors mai no tenen paràmetres. El perfil d'aquest mètode per una classe X seria:

```

X::~~X() {
    ...
}

```

Tota classe té un únic destructor. Si no està programat el destructor, el compilador proporciona un "d'ofici". En general, el destructor "d'ofici" ens va bé exceptuant si l'objecte té un o més atributs que són punters a memòria dinàmica.

El mètode destructor l'invoca **només** el sistema, just en el moment en que el bloc on es va declarar l'objecte s'acaba. Mai es cridarà explícitament el destructor.

Exemple de destructor:

```

1 class vector {
2     public:
3         ...
4         ~vector(); // destructor
5         ...
6     private:
7         int *_t; // punter de la taula d'enters
8         int _s;  // mida de la taula
9 };
10
11 vector::~~vector() {
12     delete[] _t;
13 }
14
15 void f() {
16     if ( ... ) {
17         vector v1;
18         ...
19     } // destrucció de v1 en sortir del bloc if
20
21     vector v2;
22     ...
23 } // destrucció de v2 en sortir del bloc de la funció f

```

## 1.6 Operador d'assignació

Cal tenir en compte que inicialitzar és diferent d'assignar. Quan redefinim l'operador d'assignació `=`, és similar al constructor per còpia, però l'objecte modificat (la part esquerra) és un objecte que ja existeix i retorna una referència a l'objecte destinatari de la còpia.

Tota classe té definida l'assignació. Si no la programem, el compilador proporciona l'operador d'assignació "d'ofici" que consisteix en cridar un a un l'operador d'assignació per cada atribut de l'objecte en curs. En general, l'assignació "d'ofici" ens va bé, exceptuant si l'objecte té un o més atributs que són punters a memòria dinàmica. En aquest cas cal redefinir l'operador d'assignació per tal que tingui l'efecte que nosaltres desitgem.

El perfil d'aquest mètode per una classe `X` seria:

```

X& X::operator=(const X&) {
    ...
}

```

L'operador retorna una referència a l'objecte que rep la còpia permetent així l'ús d'expressions com ara: `a = b = c;`.

Altres operadors relacionats amb el d'assignació (com ara +=, -=, etc.) acostumen a tenir el mateix perfil per la mateixa raó.

Exemple d'operador d'assignació:

```

1 class vector {
2     public:
3         ...
4         vector& operator=(const vector& v); // assignació
5         ...
6     private:
7         int *_t; // punter de la taula d'enters
8         int _s; // mida de la taula
9 };
10
11 vector& vector::operator=(const vector& v) {
12     if (&v != this) {
13         if (_s != v._s) { // si no són de la mateixa mida
14             delete[] _t; // la taula _t no ens serveix.
15             _s = v._s;
16             _t = new int[_s];
17         }
18         for (int i=0; i < _s; ++i) {
19             _t[i] = v._t[i];
20         }
21     }
22     return *this;
23 }

```

## 1.7 El component this

this@this

this és una paraula reserva en C++ i és un punter a l'objecte que invoca el mètode. És típic en C++ que l'assignació retorni la referència a l'objecte modificat de manera que, p.e., `a = b = 0` tingui sentit.

```
a = b = 0 ;
```

```
// equival a:
```

```
a = (b = 0);
```

```
// equival a:
```

```
a.operator=(b.operator=(0));
```

## 1.8 Exemple pila

### 1.8.1 Especificació de la classe pila

Fitxer `pila.hpp`:

```

1 #ifndef _PILA_HPP
2 #define _PILA_HPP
3
4 class pila {
5     public:
6         pila ();                // constructor
7
8         // tres grans
9         pila(const pila& p);    // constructor per còpia
10        ~pila ();               // destructor
11        pila& operator=(const pila& p); // operador assignació
12
13        void apilar(int x);
14        void desapilar();
15        int cim() const;
16        bool es_buida() const;
17
18    private:
19        struct node { // definició de tipus privat
20            node* seg; // punter al següent 'node'
21            int info;
22        };
23
24        node* _cim; // la pila consisteix en un punter al node del cim
25
26        // mètode privat de classe per alliberar memòria; allibera la cadena de
27        // nodes que s'inicia en el node n.
28        static void esborra_pila(node* n);
29
30        // mètode privat de classe per realitzar còpies; còpia tota la cadena de nodes
31        // a partir del node apuntat per origen i retorna un punter al node inicial de
32        // la còpia; la paraula reservada const indica que no es pot modificar el valor
33        // apuntat pel punter origen.
34        static node* copia_pila(const node* origen);
35    };
36 #endif

```

### 1.8.2 Implementació de la classe pila

Fitxer `pila.cpp`:

```

1 #include "pila.hpp"
2
3 // -----
4 // MÈTODES PRIVATS DE CLASSE
5 // -----
6
7 void pila::esborra_pila(node* n) {
8     if (n != NULL) {
9         esborra_pila(n->seg); // p->seg equival a (*p).seg
10        delete n; // allibera la memòria de l'objecte apuntat per n.
11    }
12 }
13
14 // és necessari posar pila::node com tipus del resultat per què node
15 // està definit de la classe pila
16 pila::node* pila::copia_pila(const node* origen) {
17     node* desti = NULL
18     if (origen != NULL) {
19         desti = new node;
20         desti->info = origen->info;
21
22         // copia la resta de la cadena
23         desti->seg = copia_pila(origen->seg);
24     }
25     return desti;
26 }
27
28 // -----
29 // MÈTODES PÚBLICS
30 // -----
31 pila::pila() : _cim(NULL) { }
32
33 // genera una còpia de la pila apuntada per 'p._cim'
34 pila::pila(const pila& p) {
35     _cim = copia_pila(p._cim);
36 }
37
38 // allibera la memòria de la pila apuntada per '_cim'
39 pila::~pila() {
40     esborra_pila(_cim);
41 }
42
43 pila& pila::operator=(const pila& p) {
44     if (this != &p) {
45         node* aux = copia_pila(p._cim);

```

```

46     esborra_pila(_cim);
47     _cim = aux;
48 }
49 return *this; // retorna una referència a la pila que invoca el mètode.
50 }
51
52 void pila::apilar(int x) {
53     node* n = new node;
54     n->info = x;
55     n->seg = _cim; // connecta el nou node amb el primer node de la pila i
56                  // fa que aquest sigui el cim
57     _cim = n;
58 }
59
60 void pila::desapilar() {
61     node* n = _cim;
62     if (_cim != NULL) {
63         _cim = _cim->seg;
64         delete n;
65     }
66     // faltaria tractar l'error de pila buida
67 }
68
69 int pila::cim() const {
70     if (_cim != NULL) {
71         return _cim->info;
72     }
73     // faltaria tractar l'error de pila buida
74 }
75
76 bool pila::es_buida() const {
77     return _cim == NULL;
78 }

```

### 1.8.3 Programa d'exemple que usa la classe pila

Fitxer exemple\_pila.cpp:

```

1 #include <iostream>
2 #include "pila.hpp"
3
4 using namespace std;
5
6 int main() {
7     int el;
8     pila p;

```

```

9
10     cin >> el;
11     while (el != 0) { // mentre es vagin introduïnt enters
12         p.apilar(el); // diferents de 0, apilar-los.
13         cin >> el;
14     }
15     pila q = p; // inicialització per còpia
16     while (not q.es_buida()) { // fem un palíndrom en p.
17         p.apilar(q.cim());
18         q.desapilar();
19     }
20     while (not p.es_buida()) { // imprimim i buidem p.
21         cout << p.cim(); << ' ';
22         p.desapilar();
23     }
24     cout << endl;
25 }

```

## 1.9 Exercici

L'objectiu d'aquesta exercici és implementar la classe `cj_enters` que ens permet representar i manipular conjunts d'enters (sense una restricció de mida màxima del conjunt).

La declaració d'aquesta classe amb els mètodes que heu d'implementar la trobareu en el fitxer `cj_enters.hpp` que es descriu amb detall en la següent subsecció.

Els passos a seguir són els següents:

1. trobeu una representació adequada pels objectes de la classe i escriviu els atributs necessaris en la part `private` de `cj_enters.hpp`. La representació que heu d'utilitzar és la d'una llista enllaçada dinàmica.
2. comenceu amb una implementació trivial (p.e., cos buit) per tots els mètodes en `cj_enters.cpp`, i després implementeu i proveu els diferents mètodes paulatinament. Els primers mètodes que hauríeu d'implementar són la constructora (obvi!), `insereix`, `conte` i `print`.
3. un cop provats aquests mètodes continueu implementant i provant tota la resta fins a tenir la classe feta.

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 A).

### 1.9.1 `cj_enters.hpp`

Un objecte de la classe `cj_enters` representa a un conjunt finit d'enters però del qual no se sap la mida màxima.

En els conjunts cada element ha de ser únic, per tant no poden haver elements repetits en un conjunt.

*Implementació:* La representació de les classes es trobarà en el fitxer capçalera `cj_enters.hpp` (a la part privada de cada classe); la implementació de la classe estarà en el fitxer `cj_enters.cpp`.

```
#ifndef _CJ_ENTERS_HPP_
#define _CJ_ENTERS_HPP_
#include <iostream>

using namespace std;

class cj_enters {

public:
```

Constructora per defecte. Crea un conjunt buit.

```
cj_enters();
```

Tres grans: constructora per còpia, destructora i operador d'assignació

```
cj_enters(const cj_enters& cj);
~cj_enters();
cj_enters& operator=(const cj_enters& cj);
```

Insereix l'enter *e* en el conjunt. Naturalment si *e* ja pertanyia al conjunt, el mètode no fa res.

```
void insereix(int e);
```

Unió, intersecció i diferència de conjunts. Operen modificant el conjunt sobre el que s'aplica el mètode, usant el segon conjunt com argument. Pe.: `a.unir(b)` fa que *a* guardi el resultat de  $A \cup B$ , on *A* i *B* són els valors originals dels objectes *a* i *b*.

```
void unir(const cj_enters& B);
void intersectar(const cj_enters& B);
void restar(const cj_enters& B);
```

Unió, intersecció i diferència de conjunts. Operen creant un nou conjunt sense modificar el conjunt sobre el que s'aplica el mètode. La suma de conjunts correspon a la unió, la resta a la diferència i el producte a la intersecció.

```
cj_enters operator+(const cj_enters& B) const;
cj_enters operator*(const cj_enters& B) const;
cj_enters operator-(const cj_enters& B) const;
```



Retorna cert si i només si  $e$  pertany al conjunt.

```
bool conte(int e) const;
```

Retornen els elements màxim i mínim del conjunt, respectivament. El seu comportament no està definit si el conjunt és buit.

```
int max() const;
int min() const;
```

Retorna el nombre d'elements (la cardinalitat) que té en aquests moments el conjunt.

```
int card() const;
```

Operadors relacionals. `==` retorna cert si i només si els dos conjunts (el paràmetre implícit i  $B$ ) contenen els mateixos elements; `!=` retorna cert si i només si els conjunts són diferents.

```
bool operator==(const cj_enters& B) const;
bool operator!=(const cj_enters& B) const;
```

Imprimeix el conjunt d'enters, ordenats en ordre ascendent, sobre el canal de sortida `os`; el format és  $[e_1 \ e_2 \ \dots e_n]$ , és a dir, amb caràcters espaiadors simples separant els elements i tancant la seqüència entre corxets.

```
void print(ostream& os) const;
```

```
private:
```

Els atributs i mètodes privats van aquí.

```
};
```

```
#endif
```

## 1.9.2 Activitats

Si ja has implementat la classe `cj_enters` llavors també tens acabades totes les activitats. Repassa-les per estar preparat per l'**Activitat número 1**. En cas contrari primer acaba d'implementar la classe `cj_enters` abans de continuar mirant les Activitats.

**NOTA:** En cada una de les diferents activitats cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

- 1.1 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode unir de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void unir(const cj_enters& cj);
```

- 1.2 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode intersectar de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void intersectar(const cj_enters& cj);
```

- 1.3 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode restar de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void restar(const cj_enters& cj);
```

- 1.4 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode `operator+` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

Pots implementar els mètodes auxiliars que creguis necessaris.

```
cj_enters operator+(const cj_enters& cj) const;
```

- 1.5 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode `operator*` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
cj_enters operator*(const cj_enters& cj) const;
```

- 1.6 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode `operator-` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
cj_enters operator-(const cj_enters& cj) const;
```

- 1.7 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

- b) Implementa el mètode `operator==` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
bool operator==(const cj_enters& cj) const;
```

1.8 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

b) Implementa el mètode `operator!=` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
bool operator!=(const cj_enters& cj) const;
```

1.9 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

b) Implementa el mètode `operator=` de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
cj_enters& operator=(const cj_enters& cj);
```

1.10 a) Escriu la representació de la classe `cj_enters` usant memòria dinàmica (llista enllaçada).

b) Implementa el constructor per còpia i el destructor de la classe `cj_enters`. Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
cj_enters(const cj_enters& cj);  
~cj_enters();
```



# 2

## Excepcions i genericitat

### 2.1 Gestió d'errors

Les excepcions són anomalies que succeixen durant l'execució d'un programa i que impedeixen que continuï la seva execució normal. Aquestes anomalies poden ser provocades per errors de l'usuari (intentar obrir un fitxer inexistent, etc.), per errors lògics (divisió per zero, un accés a una posició inexistent del vector, etc.), o bé errors del sistema o del maquinari.

El disseny de la gestió d'errors ha de realitzar-se posant molt de compte tant en el tractament normal com en l'erroni. En general, un disseny adequat consisteix en:

1. Assumir que les precondicions d'una determinada funció o mètode públic que estem implementant poden no complir-se i detectar qualsevol error en aquest sentit (llevat que per raons d'eficiència convingui no fer la comprovació corresponent).
2. Utilitzar altres funcions i mètodes de la mateixa o d'altres classes assegurant-nos que es verifiquen les seves respectives precondicions i no provoquen cap error en la mesura del possible. No s'ha d'incloure codi per propagar o tractar errors que no es produiran perquè hem fet un ús correcte de les funcions, mètodes, classes, etc.

Hi ha varies formes de tractar les excepcions:

- Ignorar-les (òbviament no és la millor forma).
- Avortar el programa.
- Usar codis d'error.
- Usar assercions.
- Usar el mecanisme de tractament d'excepcions del llenguatge C++.

### 2.1.1 Avortar el programa

Acabar immediatament l'execució del programa és un tractament d'errors simple però que permet com a mínim evitar obtenir resultats incorrectes.

Per fer això es pot utilitzar les funcions `exit()` o `abort()` que estan incloses a la biblioteca `<cstdlib>`.

```
1 #include <cstdlib>
2
3 int main () {
4     ...
5     if (n <= 0) {
6         cerr << "Missatge d'error" << endl;
7         exit(1);
8     }
9     ...
10 }
```

### 2.1.2 Codis d'error

Els codis d'error consisteixen en interrompre l'execució d'una funció per delegar el problema (mitjançant un codi d'error, possiblement amb informació addicional) a la funció que l'ha invocat. Aquesta funció a la seva vegada també pot delegar el problema i així successivament.

```
1 int f () {
2     int res, ret;
3     res = f1();
4     if (res == OK) {
5         res = f2();
6         if (res == OK) {
7             ret = OK;
8         }
9         else {
10            ret = ERROR2;
11        }
12    }
13    else {
14        ret = ERROR1;
15    }
16    return ret;
17 }
```

Els codis d'error obliguen a estendre la signatura de les funcions (a vegades retornant un codi d'error o afegint un paràmetre extra d'entrada i sortida) per poder passar la informació sobre l'error entre les funcions. A més cal fer un tractament dels errors de manera interna en les funcions afegint més complexitat.

### 2.1.3 Assertions

La instrucció `assert()` (de fet és una macro que està a la biblioteca `<cassert>`) comprova una condició donada, interrompent l'execució del programa i imprimint un missatge d'error si la condició no es compleix.

```
assert(n > 0); // si n <= 0 crida a abort()
```

Les assertions donen escassa informació sobre els tipus d'error que s'han produït:

- No permeten corregir situacions d'error.
- No alliberen els recursos compartits.
- Avorten l'execució del programa.

Però per altra banda, com a mínim permeten evitar l'obtenció de resultats incorrectes. En general, s'utilitzen per la comprovació de precondicions i postcondicions.

## 2.2 Excepcions en C++

Una excepció és un objecte que pot ser llançat, per més tard ser capturat i tractat. El llenguatge C++ incorpora un mecanisme unificat de tractament d'excepcions. Permet tractar les excepcions com objectes d'una classe determinada.

El mecanisme d'excepcions és molt convenient ja que permet dividir el programa en seccions separades per tractar les situacions normals i excepcionals.

### FASES DE LA GESTIÓ D'ERRORS

La gestió d'una situació anòmla en un programa té tres fases: detecció, propagació i tractament. Un cop s'hagi **detectat** un error en un punt del programa s'ha d'indicar la presència del mateix generant una excepció. Aquesta informació s'ha de fer arribar (**propagar**) fins el punt en que es procedirà a actuar (**tractament**).

#### 2.2.1 Detecció i generació d'una excepció

Una vegada hem detectat la situació d'error farem servir la instrucció `throw expressió` per generar una excepció llançant un objecte de la classe que avalua l'expressió.

```
1 // Llança una excepció de tipus enter. En concret el número 15.  
2 throw 15;  
3  
4 // Llança una excepció de tipus string.  
5 // En concret el missatge "Error del Programa".  
6 throw "Error del Programa";
```

Aquesta instrucció s’ha d’executar:

- Dins d’un bloc `try`,
- O bé dins d’una funció que hagi estat invocada dins d’un bloc `try`.
- O dins d’una funció que invoca a una funció que ha estat invocada dins un bloc `try`,
- etc.

El flux de l’execució s’interromp tan aviat es llança l’excepció. L’excepció “salta” immediatament al final de la funció on es produeix, d’allà a la funció que va fer la crida, i d’allà va saltant desfent la seqüència de crides fins que troba el codi disposat a capturar i tractar l’excepció que hem generat. És a dir, la seqüència de “salts” acaba quan l’excepció arriba a un bloc `try`. Cada cop que es desfa una crida s’invoquen els destructors apropiats (es destrueixen totes les variables locals).

### 2.2.2 Tractament de les excepcions

Un bloc `try { ... }` conté instruccions que poden ocasionar la generació d’una excepció que es vol controlar.

Un bloc `catch (tipus) { ... }` realitza el tractament de les excepcions del mateix tipus que el tipus indicat. Aquest bloc ha d’anar immediatament a continuació d’un bloc `try`.

```

1 try {
2     // bloc d'instruccions crítiques
3 }
4 catch (tipus_excepció1 var1) {
5     // gestor 1
6 }
7 catch (tipus_excepció2 var2) {
8     ...
9 }
```

Si entre els parèntesis posem tres punts llavors aquest bloc `catch` capturarà qualsevol tipus d’excepció, sigui del tipus que sigui.

```

1 try {
2     // codi que pot generar excepcions de la classes el_meu_error i d'altres.
3     ...
4 }
5 catch (el_meu_error e) {
6     // codi de tractament per excepcions de la classe el_meu_error;
7     // dins d'aquest bloc l'excepció pren el nom e.
8     ...
9 }
```



```

9  }
10 catch (std::range_error) {
11     cerr << "fora_de_rang" << endl;
12 }
13 catch (std::bad_alloc) {
14     cerr << "no_hi_ha_memòria_disponible" << endl;
15 }
16 catch (...) {
17     // codi de tractament per qualsevol altra classe d'excepció.
18     ...
19 }

```

Si una excepció que ha estat llançada no troba un gestor apropiat, es cridarà a la funció `terminate()`, la qual per defecte realitza un `abort()`.

Una excepció pot ser capturada per un `catch`, tractada i relançada per ser recollida per un bloc `try` més extern. Per relançar una excepció només cal cridar a la instrucció `throw` sense passar-li cap paràmetre.

```

1 void f(int x) {
2     try {
3         ...
4     }
5     catch (error e) {
6         ... // es tracta parcialment l'error
7         throw; // relancem l'error
8     }
9     ...
10 }
11
12 int main() {
13     try {
14         ...
15         f(x);
16         ...
17     }
18     catch (error e) {
19         ...
20     }
21 }

```

El que no s'ha de fer mai és recollir una excepció per a continuació sense fer cap tractament relançar-la. Per què simplement no es deixa que es propagui l'excepció ella sola?

```

1 void f(int x) {
2     ...

```

```

3     try {
4         g(y);
5     }
6     catch (error e) {
7         throw; // rellancem l'error. REDUNDANT!!
8     }
9     ...
10 }
11
12 int main() {
13     try {
14         ...
15         f(x);
16         ...
17     }
18     catch (error e) {
19         ...
20     }
21 }

```

### 2.2.3 Propagar una excepció

L'estàndard ANSI C++ permet, i recomana, l'ús d'especificacions d'excepcions. Les especificacions d'excepcions permeten incloure a la capçalera de cada mètode quin tipus d'excepció pot generar o propagar aquest mètode. Per fer això usarem la instrucció `throw` al final de la capçalera del mètode.

```

1 // L'acció proc1 pot propagar excepcions sols de tipus int.
2 void proc1 (int a) throw(int) { ... }
3
4 // La funció proc2 pot propagar excepcions de tipus int i char.
5 int proc2 (int a) throw(int, char) { ... }
6
7 // La funció proc3 NO pot propagar cap tipus d'excepció.
8 int proc3 (int a) throw() const { ... }

```

L'absència d'una especificació d'excepcions significa que la funció pot llançar o propagar qualsevol excepció. Aquest conveni pot semblar una mica estrany però obeeix a la compatibilitat amb versions anteriors de C++.

```

1 // L'acció proc4 pot propagar qualsevol tipus d'excepció.
2 void proc4 (int a) { ... }

```

Cal recordar que l'especificació d'excepcions serveix per indicar tant la generació com la propagació. Dins del codi d'una funció no cal que hi hagi forçosament un `throw`, potser qui genera l'excepció és una altra funció que es crida dins.

```

1 void proc5 (int a) throw(int) {
2     proc1(a);
3 }

```

## 2.3 Classe error

Una de les classes que s'utilitzen en tots els exercicis i exemples de l'assignatura és la classe error. Per tal de simplificar la gestió d'errors tots els mètodes i funcions de qualsevol classe que poden provocar una excepció llancen errors (exceptuant les classes estàndard com ara `list`, `vector` o `string`). Un objecte de la classe error és una tupla amb dos strings (el nom del mòdul o classe on es produeix l'error i el missatge d'error) i un enter (el codi d'error).

La classe ofereix les operacions:

- `modul`, `mensaje` i `codigo`: per consultar cadascun d'aquests camps per separat.
- `print`: per mostrar la informació de l'error per l'`ostream` indicat.
- `operator<<`: per facilitar mostrar informació de l'error.

Òbviament, l'ús de la classe error ha d'estar lliure d'excepcions ja que d'una altra manera entràriem en un bucle sense fi.



En els exercicis de programació de l'assignatura per provar les vostres classes disposeu de *drivers*. Els *drivers* són programes principals "avançats" que permeten testejar les classes de manera més acurada. Els drivers internament utilitzen la classe `gen_driver` i carreguen a l'inici un fitxer amb els errors. Per això, **tots els errors poden crear-se indicant únicament el seu codi.**

Tant el nom del mòdul o classe com el missatge d'error poden deixar-se buits si s'utilitza la classe `gen_driver` i es carrega inicialment un fitxer amb els errors. El nom del mòdul i el missatge associat s'obtiniran de la informació d'aquest fitxer d'errors. Una avantatge clar és que així podem modificar amb molta més facilitat els missatges d'error (per exemple, per produir-los en diferents idiomes). En els exercicis de l'assignatura se us proporcionarà el fitxer d'errors a usar. Per exemple:

```

1 // programa principal
2 ...
3 gen_driver dr("fitxer_errors.txt");
4 ...

```

```

1 // pila.cpp
2 if (...) {
3     throw error(PilaBuida);
4 }
5
6 // si el fitxer d'errors no estigués carregat hauríem de fer:
7 if (...) {
8     throw error(PilaBuida, "pila", "La_pila_és_buida");
9 }

1 // fitxer_errors.txt
2 ...
3 21 llista Element inexistent
4 31 pila La pila és buida
5 32 pila La pila és plena
6 ...

```

La captura i tractament propiament dit dels errors es farà en el mòdul principal, en aquells exercicis on se us demani. Les classes bàsicament no tractaran els errors, només els llançaran o els propagaran. En els exercicis sempre s'utilitzarà el conveni d'imprimir la informació dels errors pel canal de sortida estàndard (`cout`) i no pel d'error (`cerr`).

Per tal de no utilitzar directament els codis d'error és interessant usar constants simbòliques per gestionar els errors de cada classe.

```

1 // pila.hpp
2 #include <string>
3 #include <string/error>
4
5 class pila {
6     public:
7         // constants simbòliques per gestionar els errors
8         static const char nom_mod[] = "pila";
9
10        static const int PilaBuida = 31;
11        static const int PilaPlena = 32;
12
13        ...
14        pila(int max_elems = 10);
15        ...
16 };

1 // pila.cpp
2 #include "pila.hpp"
3 ...
4 void pila::apilar(int x) throw(error) {

```

```

5   if (_cim == max_elems) {
6       throw error(PilaPlena);
7   }
8   ...
9 }

```

### CONVENI DE PROPAGACIÓ D'ERRORS

Un conveni especial que s'usarà als exercicis de l'assignatura és el que fa referència al comportament en cas de que es produeixin varis errors simultàniament. La regla és simple: si es una invocació concreta d'una funció es produeixen varis errors simultàniament, l'excepció llançada o propagada ha de ser aquella que tingui el codi d'error menor.

## 2.4 Genericitat

S'entèn per *programació genèrica* un estil de programació en la que els algorismes són el més independents possibles dels tipus de dades que els operen. La programació genèrica en C++ té el seu fonament en les *plantilles* (anglès: *templates*) i en l'explotació sistemàtica del concepte d'iterador.

L'aplicació més visible d'aquestes tècniques és la potent *Standard Template Library* (STL), que presentarem en la següent sessió.

Un *template* permet escriure una funció o una classe en la que intervenen tipus no especificats, que en ser usada s'instanciarà amb els tipus adequats.

### 2.4.1 Plantilles de funcions

Ara veurem un parell d'exemples de com fer una plantilla d'una funció.

En el primer exemple, es vol implementar una funció que calculi el màxim per diferents tipus:

```

1 // màxim de dos enters
2 int max(int a, int b) {
3     return a > b ? a : b;
4 }
5
6 // màxim de dos reals
7 float max(float a, float b) {
8     return a > b ? a : b;
9 }
10
11 // màxim de dos caràcters
12 char max(char a, char b) {
13     return a > b ? a : b;
14 }

```

Com es pot veure el codi de les diferents funcions és exactament el mateix . Per això, per calcular el màxim de qualsevol parell de dades de tipus T es pot usar *templates*:

```

1 // plantilla màxim de dos elements
2 template<typename T>
3 T max(T a, T b) {
4     return a > b ? a : b;
5 }

```

Però, seria millor passar els paràmetres per referència per evitar les còpies dels objectes en el pas de paràmetres. Tenint en compte això un possible programa principal que calculi el màxim de diferents tipus:

```

1 // plantilla màxim de dos elements
2 template<typename T>
3 T max(const T& a, const T& b) {
4     return a > b ? a : b;
5 }
6
7 int main() {
8     double d1 = 3.5, d2 = 4.2;
9     cout << max(d1, d2);
10
11     int i1 = 3, i2 = 1;
12     cout << max(i1, i2);
13
14     ...
15 }

```

La funció genèrica max es pot aplicar sobre qualsevol tipus que tingui definit l'operador >.

Un segon exemple de plantilla de funció és una funció que implementa l'algorisme d'inserció ordenada.

```

1 // plantilla màxim de dos elements
2 template<typename T>
3 void insertion_sort(T a[], int n) {
4     for (int j = 1; j < n; ++j) {
5         T key = a[j];
6         int i = j - 1;
7         while (i >= 0 and a[i] > key) {
8             a[i+1] = a[i];
9             --i;
10        }
11        a[i+1] = key;
12    }
13 }

```

Per poder aplicar aquesta plantilla sobre un tipus *T*, cal que *T* tingui disponibles les operacions següents:

- Constructor per còpia: *T* (`const T&`)
- Assignació: *T&* `operator=(const T&)`
- Comparació: `bool operator>(const T&)`

### 2.4.2 Plantilles de classes

Definir una classe genèrica mitjançant *templates* és tan simple com definir una altra classe, encara que la sintaxi és un pèl més molesta. Per exemple, si es vol implementar una classe genèrica de piles d'elements definirem la classe `pila<Elem>` on *Elem* és el paràmetre "formal" que de la classe genèrica. Aquest paràmetre després s'instanciarà amb el tipus que necessitem en cada moment.

```

1 // pila.hpp
2 template <class Elem>
3 class pila {
4     public:
5         pila ();
6
7         pila(const pila& p);
8         pila& operator=(const pila &p);
9         ~pila ();
10        ...
11        void apilar(const Elem& x);
12        ...
13    private:
14        struct node { // En realitat seria node<Elem>
15            node* seg;
16            Elem info;
17        };
18        node* _cim;
19        int _nelems;
20 };

```

La primera línia (`template ...`) indica que en la declaració de la classe cada aparició d'*Elem* és en realitat un paràmetre formal que serà substituït per un tipus real a l'instanciar-se la classe. Un exemple d'instanciació de classe seria:

```

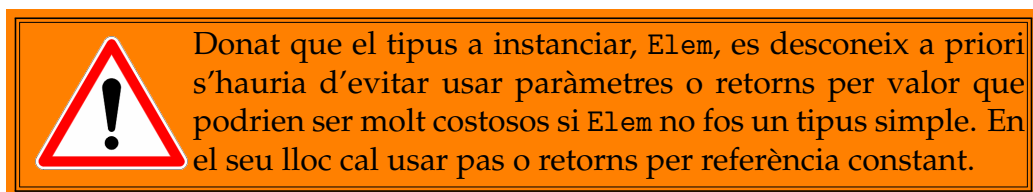
1 #include "pila.hpp" // defineix la classe genèrica pila<Elem>
2 ...
3 pila<char> p; // p és una pila de caràcters
4 pila<string> q; // q és una pila d'strings
5 pila<int*> r; // r és una pila de punters a int

```

```

6
7 // s és una pila de piles d'enters.
8 // l'espai entre <int> i > és imprescindible per tal que el compilador
9 // no ho confongui amb l'operador >>.
10 pila<pila<int> > s;

```



La implementació d'una classe genèrica segueix la mateixa pauta: cal anteposar la declaració `template ...` en la definició de cada mètode:

```

1 template <class Elem>
2 pila<Elem>::pila () {
3     _cim = NULL;
4     _nelems = 0;
5 }
6
7 template <class Elem>
8 void pila<Elem>::apilar(const Elem& x) {
9     node* p = new node;
10    p->seg = _cim;
11    p->info = x;
12    _cim = p;
13    ++_nelems;
14 }
15
16 ...

```

Hi ha dos punts interessants a comentar. Per un costat, el nom de la classe no és `pila`, sinó `pila<Elem>`. Per això hem de definir `pila<Elem>::apilar` i no `pila::apilar`. Un cop ja hem donat la “pista” al compilador de que estem definint un mètode de la classe `pila<Elem>` ja podem ometre en la resta `<Elem>` i escriure, per exemple:

```

template <class Elem>
pila<Elem>& pila<Elem>::operator=(const pila& p) {
    ...
}

```

en comptes de:



```

template <class Elem>
pila<Elem>& pila<Elem>::operator=(const pila<Elem>& p) {
    ...
}

```

Per una altra banda, s'ha d'estar pendent de les suposicions que es fan al respecte sobre el tipus `Elem`. Per exemple, la constructora per defecte (d'ofici) per la classe `node` cridarà a la constructora per defecte d'`Elem` per l'atribut `info`. Així doncs, les classes amb que instanciem `Elem` han de tenir constructora per defecte. Recordeu que si definim una constructora amb paràmetres per una classe, llavors no hi haurà constructora per defecte per aquesta classe. Anàlogament, al destruir un `node` s'invocarà la destructora d'`Elem` i per tant és necessari que existeixi. Finalment, en l'assignació `p -> info = x;` que es fa a apilar, estem usant l'operador d'assignació entre `Elms`, així doncs aquest també ha d'estar definit. En altres casos caldrà que s'hagi definit la igualtat, els operadors de comparació, el constructor per còpia, etc.

### IMPLEMENTACIÓ D'UNA CLASSE GENÈRICA

Cal remarcar que les classes genèriques en C++ (`template`) tenen la particularitat que la implementació no es posa en el fitxer `.cpp` sinó en un altre fitxer que anomenarem `.t`. Això ho fem per indicar que el fitxer que conté la implementació no s'ha de compilar per separat i que s'estan usant *templates*. Recordeu que en el cas de les classes genèriques el fitxer que conté la implementació no inclou MAI al fitxer capçalera (`.hpp`).

Un problema del mecanisme de *templates* és que actualment no hi ha cap compilador de C++ que admeti la seva compilació separada, per tant el codi que implementa una classe o funció genèrica ha d'estar en la mateixa unitat de compilació que el codi que instancia i usa el codi genèric.

Per tal de "simular" el model de separació d'especificació i implementació que hem utilitzat fins ara amb les classes no genèriques adoptarem el conveni següent: el mòdul que usa la classe genèrica inclou al fitxer `.hpp` corresponent; el fitxer `.hpp` inclou al fitxer que conté la implementació de la classe i/o funcions genèriques. El fitxer que conté la implementació de la classe genèrica no s'ha de compilar per separat, per això li donarem l'extensió `.t` en comptes de l'habitual `.cpp`.

Per obtenir més informació sobre la compilació de les classes genèriques veure la secció B.1.2.

```

1 // programa que usa la classe genèrica pila<Elem>
2 #include "pila.hpp"
3 ...
4 pila<int> p;
5 ...

```

```

1 // pila.hpp
2 template <class Elem>
3 class pila {
4     public:
5         pila ();
6
7         pila(const pila& p);
8         pila& operator=(const pila &p);
9         ~pila ();
10        ...
11    private:
12        ...
13 };
14 #include "pila.t"

1 // pila.t
2
3 // No pot aparèixer #include "pila.hpp" ja que sinó tindríem un bucle infinit
4 // d'inclusions.
5
6 template <class Elem>
7 pila<Elem>::pila () {
8     _cim = NULL;
9     _nelems = 0;
10 }
11 ...

```

## 2.5 Exercici

En aquest exercici desenvolupareu la classe `conjunt<T>`, és a dir, una classe genèrica on el tipus dels elements del conjunt pot ser qualsevol. S'assumeix que el tipus `T` té definit el constructor per còpia, el destructor i l'operador `<`. També suposarem que existeix l'operador `<<` per imprimir objectes de la classe `T` sobre un canal de sortida, per exemple, `cout`.

En general n'hi haurà prou amb modificar lleugerament la implementació realitzada en l'exercici anterior per tal que en comptes de gestionar enters es manegin `T`'s. S'ha canviat l'especificació d'algunes operacions per tal que els paràmetres d'entrada siguin `const T&` ja que en general serà més eficient el pas d'una referència constant que un pas per valor (quan es tractava de `int`'s això no era rellevant). També s'han afegit especificacions d'error; en general, diversos mètodes declaren que propaguen excepcions de la classe `error` perquè pot ocórrer que no hi hagi prou memòria dinàmica (i llavors `new` llança una excepció).

La implementació s'ha d'escriure en un fitxer `conjunt.t` que s'inclou a través del fitxer `conjunt.hpp`.

Proveu la classe genèrica substituint `cj_enters` per `conjunt<int>` en els programes d'exemple d'exercicis anteriors. Proveu el funcionament amb `conjunt<string>` (cal posar `#include <string>` per poder utilitzar la classe `string` en el programa).

En el montatge (link) s'ha d'utilitzar la biblioteca `libesin`:

```
g++ -o executable fich1.o ... -lesin
```

**ALERTA!!** Llegir amb atenció l'explicació inicial de la classe al respecte dels costos màxims de les operacions.

### 2.5.1 conjunt.hpp

Un objecte de la classe `conjunt` representa a un conjunt finit de `T`'s però del qual no se sap la mida.

En els conjunts cada element ha de ser únic, per tant no poden haver elements repetits en un conjunt.

*Decisions sobre les dades:* S'assumeix que la classe `T` té destructora, constructora per còpia i l'operador `<`. També s'assumeix que està definit l'operador `<<` que permet escriure un `T` en un ostream.

*Implementació:* La representació de les classes es trobarà en el fitxer capçalera `conjunt.hpp` (a la part privada de cada classe); la implementació de la classe estarà en el fitxer `conjunt.t`.

Les tres grans, els mètodes `insereix`, `conte`, l'operador `==` i l'operador `!=` han de tenir cost menor o igual que  $n$  en el cas mig. Els mètodes `card`, `max` i `min` han de tenir cost constant. El mètode `interseccionar` i l'operador `*` han de tenir cost menor que  $n^2$  i la resta de mètodes cost menor o igual que  $n^2$  (sent  $n$  en tots els casos el nombre d'elements del conjunt).

```
#ifndef _CONJUNT_HPP_
#define _CONJUNT_HPP_
#include <iostream>
#include <esin/error>
```

```
using namespace std;
```

```
template <class T>
class conjunt {
```

```
public:
```

Constructora per defecte. Crea un conjunt buit.

$Cost = \Theta(1)$ .

```
conjunt() throw(error);
```

Tres grans: constructora per còpia, destructora i operador d'assignació.

$Cost \leq \Theta(n)$  on  $n$  és el nombre d'elements del conjunt.

```
conjunt(const conjunt& cj) throw(error);
conjunt& operator=(const conjunt& cj) throw(error);
~conjunt() throw();
```

Insereix l'enter  $e$  en el conjunt. Naturalment si  $e$  ja pertanyia al conjunt, el mètode no fa res.  
 $Cost \leq \Theta(n)$  on  $n$  és el nombre d'elements del conjunt.

```
void insereix(const T& x) throw(error);
```

Unió, intersecció i diferència de conjunts. Operen modificant el conjunt sobre el que s'aplica el mètode, usant el segon conjunt com argument. P.e.:  $a.unir(b)$  fa que  $a$  guardi el resultat de  $A \cup B$ , on  $A$  i  $B$  són els valors originals dels objectes  $a$  i  $b$ .  
 $Cost$  d'unir i restar  $\leq \Theta(n^2)$  i d'interseccionar  $< \Theta(n^2)$  on  $n$  és el nombre d'elements del conjunt.

```
void unir(const conjunt& B) throw(error);
void interseccionar(const conjunt& B) throw(error);
void restar(const conjunt& B) throw(error);
```

Unió, intersecció i diferència de conjunts. Operen creant un nou conjunt sense modificar el conjunt sobre el que s'aplica el mètode. La suma de conjunts correspon a la unió, la resta a la diferència i el producte a la intersecció.  
 $Cost$  de l'operator+ i l'operator-  $\leq \Theta(n^2)$  i de l'operator\*  $< \Theta(n^2)$  on  $n$  és el nombre d'elements del conjunt.

```
conjunt operator+(const conjunt& B) const throw(error);
conjunt operator*(const conjunt& B) const throw(error);
conjunt operator-(const conjunt& B) const throw(error);
```

Retorna cert si i només si  $e$  pertany al conjunt.  
 $Cost \leq \Theta(n)$  on  $n$  és el nombre d'elements del conjunt.

```
bool conte(const T& x) const throw();
```

Retornen els elements màxim i mínim del conjunt, respectivament. Es llança una excepció de la classe error (definida en <sin/error>) si el conjunt és buit.  
 $Cost = \Theta(1)$ .

```
T max() const throw(error);
T min() const throw(error);
```

Retorna el nombre d'elements (la cardinalitat) del conjunt.  
 $Cost = \Theta(1)$ .

```
int card() const throw();
```

Operadors relacionals. `==` retorna cert si i només si els dos conjunts (el conjunt sobre el que aplica i *B*) contenen els mateixos elements; `!=` retorna cert si i només si els conjunts són diferents.  $Cost \leq \Theta(n)$  on  $n$  és el nombre d'elements del conjunt.

```
bool operator==(const conjunt& B) const throw();
bool operator!=(const conjunt& B) const throw();
```

Imprimeix el conjunt d'enters, ordenats en ordre ascendent, sobre el canal de sortida *os*; el format és  $[e_1 \ e_2 \ \dots e_n]$ , és a dir, amb caràcters espaiadors simples separant els elements i tancant la seqüència entre corxets.

$Cost \leq \Theta(n)$  on  $n$  és el nombre d'elements del conjunt.

```
void print(ostream& os) const throw();
```

Gestió d'errors.

```
static const int NoMinMaxEnConjBuit = 10;
```

```
private:
```

Els atributs i mètodes privats van aquí.

```
};
```

```
#include "conjunt.t"
```

```
#endif
```

## 2.5.2 Activitats

Si ja has implementat la classe *conjunt* llavors també tens acabades totes les activitats. Repassa-les per estar preparat per l'**Activitat número 2**. En cas contrari primer acaba d'implementar la classe *conjunt* abans de continuar mirant les Activitats.

Revisa que la teva implementació compleix amb els costos indicats.

**NOTA:** En cada una de les diferents activitats cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

2.1 a) Escriu la representació de la classe *conjunt* usant memòria dinàmica.

b) Implementa el mètode *unir* de la classe *conjunt* amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void unir(const conjunt& cj) throw(error);
```

2.2 a) Escriu la representació de la classe *conjunt* usant memòria dinàmica.

- b) Implementa el mètode `intersectar` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void intersectar(const conjunt& cj) throw(error);
```

- 2.3 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `restar` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
void restar(const conjunt& cj) throw(error);
```

- 2.4 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `operator+` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
conjunt operator+(const conjunt& cj) const throw(error);
```

- 2.5 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `operator*` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
conjunt operator*(const conjunt& cj) const throw(error);
```

- 2.6 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `operator-` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
conjunt operator-(const conjunt& cj) const throw(error);
```

- 2.7 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `operator==` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
bool operator==(const conjunt& cj) const throw();
```

- 2.8 a) Escriu la representació de la classe `conjunt` usant memòria dinàmica.

- b) Implementa el mètode `operator!=` de la classe `conjunt` amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
bool operator!=(const conjunt& cj) const throw();
```

- 2.9 a) Escriu la representació de la classe conjunt usant memòria dinàmica.
- b) Implementa el mètode `operator=` de la classe conjunt amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
conjunt& operator=(const conjunt& cj) throw(error);
```

- 2.10 a) Escriu la representació de la classe conjunt usant memòria dinàmica.
- b) Implementa el constructor per còpia i el destructor de la classe conjunt amb cost menor que  $\Theta(n^2)$ . Cal que escriguis tots els mètodes addicionals que usis explícitament en la teva implementació.

```
conjunt(const conjunt& cj) throw(error);  
~conjunt() throw();
```





# 3

## La biblioteca estàndard de C++

### 3.1 Introducció a l'STL

Amb freqüència hem de gestionar estructures de dades que són col·leccions d'objectes o elements. A aquest tipus d'estructures de dades se les anomena *contenidors* (*containers*). Exemples coneguts són les piles, les llistes, els conjunts, els diccionaris, les cues de prioritat, etc.

La biblioteca estàndard de C++ ofereix una gran varietat de classes i funcions útils. Entre elles hem de destacar els *stream's* per entrada/sortida, i els *string's*. A més inclou l'anomenada **STL** (Standard Template Library) que defineix diverses classes genèriques denominades contenidores, i algorismes sobre aquestes classes. Algunes d'aquestes classes contenidores són:

- **Vectors:** `vector<T>`
  - Seqüència de mida variable i dinàmica.
  - Accés aleatori (podem accedir a qualsevol element de la seqüència). Accedir als elements per la posició del seu índex té cost constant.
  - Les insercions i els esborrats pel final tenen cost constant.
- **Dobles cues:** `deque<T>` (**double-ended queue**)
  - Seqüència de mida variable i dinàmica.
  - Accés aleatori (podem accedir a qualsevol element de la seqüència).
  - Les insercions i els esborrats pel final tenen cost constant.
  - Vectors i deque's tenen una interfície molt similar i poden ser utilitzats per propòsits similats, encara que internament ambdues estructures treballen de manera molt diferent.
- **Llistes:** `list<T>`

- Seqüència de mida variable i dinàmica.
- Accés seqüencial.
- Insercions i esborrats en qualsevol posició designada per un iterador en temps constant.
- **Conjunts:** `set<K>`
  - Elements únics. Els elements mateixos són les claus.
  - Recuperació ràpida de les claus ( $O(\log(n))$ ).
- **Arrays associatius:** `map<K, I>`
  - claus úniques de tipus K amb informació associada de tipus I.
  - recuperació ràpida de la informació associada a les claus ( $O(\log(n))$ ).

A continuació veurem les classes `string`, `vector` i `list` amb més detall.

## 3.2 La classe `string`

Un *string* és una cadena de caràcters. La biblioteca estàndard de C++ ofereix una classe `string` que ens permet gestionar-los amb tota comoditat. Ocasionalment, haurem d'utilitzar cadenes de caràcters representades segons les convencions de C, és a dir, un array de caràcters (`const char*`) acabat amb el caràcter el codi ASCII del qual és 0 (`'\0'`). A aquestes últimes cadenes les anomenarem C-strings per distinguir-les dels strings que proporciona C++.

Per usar `string` haurem d'incloure el fitxer del mateix nom:

```
#include <string>
```

### 3.2.1 Ús bàsic

La classe ofereix les operacions habituals de construcció, assignació, etc. i podem llegir-los o imprimir-los d'igual forma que els tipus elementals. També podem assignar a un `string` una cadena literal entre cometes (un C-string constant). A vegades necessitem produir un C-string a partir d'un `string`, per això usarem `c_str()`:

```
1 string s;
2 cout << "Nom del fitxer: ";
3 cin >> s;
4 string t = "hola";
5 fstream f(s.c_str()); // la constructora de la classe fstream
6                       // (ver apendice B) té com paràmetre el
7                       // nom del fitxer que és un const char*
```

### 3.2.2 Operadors + i +=

Els operadors + i += serveixen per concatenar.

```

1 string s, t; // crea dos strings buits
2 s = "Hola"; // assigna el C-string constant "Hola"
3             // a l'string 's'
4
5 t = s + "┐mon!"; // concatena 's' amb "mon!"
6 cout << t << endl; // imprimeix Hola mon!
7
8 t += "┐i┐adeu!";
9 cout << t << endl; // imprimeix Hola mon! i adeu!

```

Un caràcter (char) també pot utilitzar-se com un valor del tipus, exceptuant en la inicialització:

```

1 string s = 'h'; // error!
2 string t("cola");
3 t = t + '*';    // ok!

```

### 3.2.3 Longitud

La longitud d'un string s'obté amb length() o size() i podem accedir als caràcters individualment, com si es tractés d'un array (però no ho és):

```

1 bool es_vocal(char c) { ... }
2
3 int quantes_vocals(const string& s) {
4     int nv = 0;
5     for (int i = 0; i < s.length(); ++i) {
6         if (es_vocal(s[i])) {
7             ++nv;
8         }
9     }
10    return nv;
11 }

```

### 3.2.4 Operadors de comparació

Els operadors de comparació entre strings estan definits respectant l'ordre alfabètic:

```

1 string s = "casa";
2 string t = "cama";
3 string u = "dado";

```

```

4 string v = u + "s"; // v == "dados"
5
6 cout << (s < t) << endl; // imprimeix false
7 cout << (s <= u) << endl; // imprimeix true
8 cout << (u > v) << endl; // imprimeix false
9 cout << (s != t) << endl; // imprimeix true

```

### 3.2.5 Mètode substr

El mètode `substr()` ens permet extreure *substrings* d'un string donat:

```

1 string s = "portaavions";
2 s.substr()           // retorna una copia de s
3 s.substr(5);         // retorna el substring "avions"
4 s.substr(1,6);       // retorna "ortaav"

```

En general, `substr(i, n)` retorna el substring que comença en la posició *i* i té longitud *n* (ambdós inclosos). Els caràcters s'indexen de 0 en endavant. Si *n* no es dóna, llavors retorna el substring que va des de la posició *i* fins el final. Si *i* tampoc es dóna llavors es considera que *i* = 0. Es produeix un error si *i* > `length()`.

### 3.2.6 Mètode replace

Amb `replace()` podem reemplaçar un substring per un altre. Per exemple,

```

1 string s = "portaavions";
2
3 s.replace(5,6,"helicopter");
4 cout << s << endl; // imprimeix portahelicopter

```

Existeixen versions més sofisticades de `replace`; en la versió bàsica de l'exemple donem la posició inicial i la longitud del substring a reemplaçar i l'string que reemplaça.

### 3.2.7 Mètode find

Per acabar aquesta breu descripció cal comentar algunes de las facilitats que proporciona la classe `string` per la cerca dins d'un string. El mètode bàsic s'anomena `find()` i ens permet trobar la primera ocurrència d'un caràcter o substring en un string a partir d'una certa posició. El mètode retorna la posició d'inici del substring o caràcter buscat o `npos` (un valor especial) per indicar el fracàs de la cerca.

```

1 string s = "Lola, pasame la cola";
2
3 s.find("ola");           // retorna 1
4 s.find("ola", 3);        // retorna 17
5 s.find("pase");          // retorna npos
6 s.find('p');             // retorna 7
7 s.find('l', 3);          // retorna 13

```

El primer argument de `find()` és el caràcter o substring que se busca. El segon és la posició (inclusivament) a partir de la qual s'inicia la cerca. Per defecte, la cerca s'inicia en el principi de l'string.

El conveni d'usar `npos` per indicar el fracàs d'una cerca introdueix certs inconvenients a l'hora d'usar el mètode `find()`. El resultat d'una cerca sempre s'hauria de recollir en una variable del tipus `string::size_type` i haurem de testejar si el resultat és `string::npos`.

```

1 // reemplaça totes les aparicions en s de s1 per s2
2 // ex: { s = "ana se come una banana" }
3 // global_replace(s, "ana", "alle")
4 // { s = "alle se come una ballena" }
5 void global_replace(string& s, const string& s1,
6     const string& s2) {
7     string::size_type idx = 0;
8     int l1 = s1.length();
9     int l2 = s2.length();
10    while ((idx = s.find(s1, idx)) != string::npos) {
11        s.replace(idx, l1, s2);
12        idx += l2;
13    }
14 }
```

### 3.3 vector<T>

Un objecte de la classe `vector` és conceptualment similar a un array, però sense algunes de les desavantatges dels arrays i ofereix funcionalitats addicionals, com ara que s'expandeixin i es contreguin segons la necessitat.

Un dels desavantatges de `vector` és que quan la capacitat es gestiona automàticament en general consumeix més memòria que un array. Això és degut a que reserva espai addicional d'emmagatzematge de cara a un creixement futur del vector.

Com a classe contenidora estàndard ofereix un repertori d'operacions molt extens, incloent algunes que no són típiques dels vectors (p.e. inserció i esborrat d'elements en posicions intermitges).

En tots els contenidors la classe dels elements que es vol emmagatzemar (tipus `T`) ha de tenir definida el constructor per còpia, l'assignació i l'operador d'igualtat. A més la semàntica d'aquestes tres operacions ha de ser coherent:

```

1 T a = b; // constructor per còpia
2 c = b; // operador d'assignació
3 bool ok = (a == c); // ha de ser true
4 bool ok = (a == b); // ha de ser true
```

Per usar `vector` haurem d'incloure el fitxer del mateix nom:

```
#include <vector>
```

### 3.3.1 Ús bàsic

La mida inicial d'un vector es pot indicar en el moment de la creació, però pot augmentar si s'afegeixen elements al final, p.e. mitjançant el mètode `push_back`.

Constructor de vector.

```
explicit vector();
explicit vector(size_type n, const T& value= T());
```

Tres grans.

```
vector(const vector<T>& x);
~vector();
vector<T>& operator=(const vector<T>& v);
```

**Exemple d'ús:**

```
1 #include <vector>
2
3 int main() {
4     // creem un vector de 9 enters
5     vector<int> v1(9);
6
7     // crea un vector de 10 caràcters inicialitzats amb 'a'
8     vector<char> v1(10, 'a');
9 }
```

### 3.3.2 Iteradors

Els iteradors de la classe `vector` funcionen igual que els iteradors de la classe `list`. Per més informació sobre els iteradors consultar la classe `list` (veure la secció 3.4.2).

Retorna un iterador al principi.

```
iterator begin();
const_iterator begin() const;
```

Retorna un iterador al final.

```
iterator end();
const_iterator end() const;
```

Retorna un iterador invers que es refereix a l'últim element de la llista.

```
reverse_iterator rbegin();
```

```
const_reverse_iterator rbegin() const;
```

Return un iterador invers que es refereix al primer element de la llista.

```
reverse_iterator rend();  
const_reverse_iterator rend() const;
```

### 3.3.3 Capacitat

Internament, els vectors —com tots els contenidors— tenen una mida que representa la quantitat d'elements que conté el vector. No obstant això, els vectors, també tenen una capacitat, que determina la quantitat d'espai d'emmagatzematge que tenen assignats, i que pot ser igual o superior a la real. La quantitat extra d'emmagatzematge que el vector té assignat no s'utilitza, però està reservada per al vector per ser usada en el cas que creixi. D'aquesta manera, el vector no ha de reassignar l'emmagatzematge, cada vegada que creix. Només ho farà quan aquest espai extra s'hagi esgotat i un nou element s'insereixi (que només hauria de passar en freqüència logarítmica en relació amb la seva grandària).

Les reassignacions poden ser una operació costosa en termes de rendiment, ja que generalment involucren que tot l'espai d'emmagatzematge utilitzat pel vector es copiï a una nova ubicació. Per tant, cada vegada que hi hagi previst un gran augment de la mida per a un vector, es recomana indicar explícitament una capacitat per al vector utilitzant el mètode `vector::reserve`.

Retorna la mida.

```
size_type size() const;
```

Retorna la mida màxima. Aquest màxima és la mida màxima potencial que la llista pot arribar a tenir degut a limitacions del sistema o de la biblioteca.

```
size_type max_size() const;
```

Canvia la mida, esborrant o afegint elements en cas necessari.

```
void resize(size_type sz, T c = T());
```

Retorna la mida de la capacitat d'emmagatzematge assignada.

```
size_type capacity() const;
```

Indica si el vector és buit.

```
bool empty() const;
```

Demana que la capacitat ha de ser com a mínim la indicada.

```
void reserve(size_type n);
```

### 3.3.4 Accés als elements

La classe ofereix mètodes d'accés per l'índex: o bé mitjançant l'operador d'indexació tradicional [], o bé mitjançant el mètode `at` que comprova el rang i llança una excepció en cas necessari.

Accés a l'element n-èssim.

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Accés a l'element n-èssim.

```
reference at(size_type n);
const_reference at(size_type n) const;
```

Accés al primer element.

```
reference front();
const_reference front() const;
```

Accés a l'últim element.

```
reference back();
const_reference back() const;
```

#### Exemple d'ús:

```
1 #include <vector>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     vector<int> v;
8     vector<vector<double>> > mat;
9     for (int i=0; i < 10; ++i) {
10         v.push_back(i);
11     }
12     for (int i=0; i < v.size(); ++i) {
13         cout << v[i] << " ";
14     }
15     cout << endl;
```



```

16     try {
17         for (int i=0; i < 100; ++i) {
18             cout << v.at(i) << " ";
19             // llança una excepció si la posició no existeix
20         }
21         cout << endl;
22     }
23     catch (...) {
24         cerr << "Fora de rang!" << endl;
25     }
26 }

```

### 3.3.5 Modificadors

En comparació amb els contenidors bàsics de seqüències (`deque` i `list`), els vectors són generalment els més eficients en temps per accedir als elements i per afegir o treure elements del final de la seqüència.

Per a les operacions que impliquin inserir o eliminar elements en posicions que no siguin el final, tenen pitjors resultats que `deque` i `list`, i a més els iteradors i referències són menys consistents que els de les llistes.

Afegeix nous elements a la llista, eliminant tots els elements que contenia la llista anteriorment. En concret afegeix  $n$  vegades l'element  $u$ .

```
void assign(size_type n, const T& u);
```

Afegeix un element al final.

```
void push_back(const T &x);
```

Esborra l'últim element.

```
void pop_back();
```

Insereix elements just davant de l'element apuntat per l'iterador.

```
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x );
```

Elimina elements.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

Intercanvia el contingut dels dos vectors (el vector en curs i  $v$ ).

```
void swap(vector<T> &v);
```

Esborra tot el contingut del vector.

```
void clear();
```

### 3.4 list<T>

La classe `list` ofereix operacions usals de llistes, incloent recorreguts en ambdues direccions, insercions i esborrats en punts arbitraris, etc.

La flexibilitat i eficiència d'aquesta classe resideix en les classes auxiliars d'iteradors. Un **iterador** és un "apuntador" a un element de la llista i s'usa per consultar l'element apuntat, eliminar-lo, inserir un nou element, etc.

Comparada amb els altres contenidors estàndards de seqüències (`vector` i `deque`), les llistes en general acostumen a ser millors inserint, accedint i movent elements en qualsevol posició dins del contenidor, i també en els algorismes que fan ús d'aquestes operacions, com ara els algorismes d'ordenació.

El principal inconvenient de les llistes en comparació amb els altres contenidors de seqüències és que no tenen accés directe als elements per la seva posició. Per exemple, per accedir al sisè element d'una llista s'ha de recórrer des d'una posició coneguda (com el principi o el final) fins arribar a aquesta posició, la qual cosa pren un temps lineal que depèn de la distància on es troba l'element.

Les llistes també consumeixen una mica de memòria extra per mantenir la informació que encadena els diferents elements. Aquest pot ser un factor important quan la llista és gran i els elements de mida petita.

Per usar `list` haurem d'incloure el fitxer del mateix nom:

```
#include <list>
```

#### 3.4.1 Ús bàsic

Constructor de `list`.

```
explicit list();
explicit list(size_type n, const T& value= T());
```

Tres grans.

```
list(const list<T> &l);
~list();
list<T>& operator=(const list<T>& l);
```

### 3.4.2 Iteradors

A vegades necessitem recórrer els elements d'un d'aquests contenidors. Una possibilitat és dota a la classe corresponent d'un punt d'interès i operacions que permetin desplaçar el punt d'interès i consultar l'element al qual *apunta* el punt d'interès. Una altra solució que és molt més potent i flexible consisteix en definir una o més classes auxiliars d'**iteradors** associats a la classe contenidora.

Un **iterador** abstruïu la idea d'índex o punter a un element. S'assembla molt a un punt d'interès, però tenen diferències molt importants:

- el punt d'interès està sota el control de la pròpia classe contenidora i només pot haver-hi un.
- els iteradors són externs a la classe contenidora i es poden crear tants com siguin necessaris.

Els iteradors poden ser assignats i comparats (==, !=). Els operadors d'increment i decrement desplacen l'iterador des d'un element al seu successor o predecessor. L'operador de desreferència \* aplicat a un iterador permet accedir al seu contingut. Però un iterador **NO** és un punter, encara que en molts aspectes es comporti de manera similar.

Aquesta classe està equipada també amb iteradors inversos (`reverse_iterator` i `reverse_const_iterator`) que operen al revés que els iteradors normals. En termes abstractes treballen sobre la llista invertida.

Retorna un iterador al principi.

```
iterator begin();
const_iterator begin() const;
```

Retorna un iterador al final.

```
iterator end();
const_iterator end() const;
```

Retorna un iterador invers que comença pel final.

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Retorna un iterador invers que comença pel principi.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

**Exemple d'ús:**

```

1 // Acció que donada una llista d'enters inicialitza tots els elements amb
2 // l'enter indicat.
3 void assigna_x(list<int>& l, int x) {
4     list<T>::iterator it = l.begin();
5     while (it != l.end()) {
6         *it = x;
7         ++it;
8     }
9 }
10
11 // Acció que donada una llista mostra els seus elements per l'ostream
12 // indicat.
13 // ATENCIÓ: Per recórrer aquesta llista caldrà usar un iterador constant
14 // donat que la llista que ens passen no es pot modificar.
15 template <typename T>
16 void mostrar(const list<T>& l, ostream& os) {
17     list<T>::const_iterator it = l.begin();
18     while (it != l.end()) {
19         os << *it << endl;
20         ++it;
21     }
22 }

```

### 3.4.3 Capacitat

Indica si la llista és buida.

```
bool empty() const;
```

Retorna la mida de la llista.

```
size_type size() const;
```

Retorna la mida màxima. Aquest màxima és la mida màxima potencial que la llista pot arribar a tenir degut a limitacions del sistema o de la biblioteca.

```
size_type max_size() const;
```

Canvia la mida de la llista, esborrant o afegint elements en cas necessari.

```
void resize(size_type sz, T c = T());
```

**Exemple d'ús:**

```

1 #include <iostream>
2 #include <list>
3
4 using namespace std;
5
6 int main () {
7     list<float> lst;
8     cout << lst.empty() << endl;    // true
9     cout << lst.size() << endl;    // 0
10    lst.push_back(1.0);
11    lst.push_back(2.0);
12    lst.push_back(3.0);
13    cout << lst.empty() << endl;    // false
14    cout << lst.size() << endl;    // 3
15 }

```

### 3.4.4 Accés als elements

Accés al primer element.

```

reference front();
const_reference front() const;

```

Accés a l'últim element.

```

reference back();
const_reference back() const;

```

### 3.4.5 Modificadors

La classe `list` té les operacions usuales d'inserció i esborrat pel final (`push_back`, `pop_back`) i pel principi (`push_front`, `pop_front`)

Afegeix nous elements a la llista, eliminant tots els elements que contenia la llista anteriorment. En concret afegeix  $n$  vegades l'element  $u$ .

```

void assign(size_type n, const T& u);

```

Afegeix un element al principi de la llista.

```

void push_front(const T &x);

```

Esborra el primer element de la llista.

```

void pop_front();

```

Afegeix un element al final.

```
void push_back(const T &x);
```

Esborra l'últim element de la llista.

```
void pop_back();
```

Insereix el nombre d'elements indicat abans de l'iterador.

```
iterator insert (iterator position, const T& x);  
void insert (iterator position, size_type n, const T& x);
```

Esborra de la llista l'element apuntat per l'iterador o tots els elements que estan entre els dos iteradors.

```
iterator erase (iterator position);  
iterator erase (iterator first, iterator last);
```

Intercanvia el contingut de les dues llistes (la llista en curs i *lst*).

```
void swap(list<T>& lst);
```

Esborra tot el contingut de la llista.

```
void clear();
```

### 3.4.6 Altres operacions

Algunes de funcionalitats addicionals que tenen les llistes són les següents:

Mou elements d'una llista a l'altra.

```
void splice(iterator pos, list<T>& x);  
void splice(iterator pos, list<T>& x, iterator i);  
void splice(iterator pos, list<T>& x, iterator first, iterator last);
```

Esborra els elements que tenen un valor específic.

```
void remove(const T& value);
```

Esborra els valors duplicats.

```
void unique();
```

Fusiona dues llistes. Combina  $x$  a la llista en curs, inserint tots els elements de  $x$  en les seves respectives posicions ordenades. Aquest mètode buida  $x$  i incrementa la mida de la llista.

```
void merge(list<T>& x);
```

Ordena els elements que hi ha en el contenidor del més petit fins el més gran. L'ordenació es realitza mitjançant la comparació dels elements a la llista de dos en dos utilitzant un algorisme d'ordenació. Aquesta operació no suposa la construcció o destrucció de cap objecte.

```
void sort();
```

### Exemple d'ús

```
1 #include <list>
2 #include <string>
3 #include <iostream>
4
5 int main() {
6     list<string> lst;
7     string s;
8     while (cin >> s) { // 'cin >> s' retorna 0 ≡ false si s'acaba l'input
9         lst.push_back(s);
10    }
11    lst.sort(); // ordena 'lst'
12    mostrar(lst, cout);
13 }
```

## 3.5 Exercici

A ciències de la computació i teoria de la informació, la codificació Huffman és un algorisme utilitzat per compressió de dades. El terme es refereix a l'ús d'una taula de codis de longitud variable per a codificar un determinat símbol (com pot ser un caràcter en un fitxer), on la taula ha estat emplenada d'una manera específica en base a la probabilitat estimada d'aparició de cada possible valor d'aquest símbol.

L'exercici principal d'aquesta secció consisteix en implementar:

- la classe `arbbin` de la subsecció 3.5.2, en la que apareix llistat el fitxer `arbbin.hpp`.
- el mòdul `huffman` de la subsecció 3.5.3, en la que apareix llistat el fitxer `huffman.hpp`.

En cadascuna d'aquestes subseccions apareix també l'especificació de cadascun dels mètodes de la classe/mòdul.

Per implementar aquesta classe es recomana seguir els següents passos:

1. Decidiu la representació interna d'un arbre binari genèric.
2. Implementeu la constructora de la classe `arbbin` i proveu-la.

3. Implementeu la classe `arbbin::iterador` i proveu-la.
4. Acabeu de completar la classe `arbbin` i proveu-la.
5. Implementeu el mòdul `huffman` i proveu que funciona correctament.

### 3.5.1 Algorisme de Huffman

Aquesta versió de l'algorisme de Huffman està optimitzada per seqüències d'enters positius  $S$  de longitud  $\geq 2$ , on tots els elements de  $S$  estan en un rang  $1 \dots k$  i cadascun dels valors entre 1 i  $k$  apareix com a mínim un cop a la seqüència.

Donada una seqüència d'elements primerament es comptabilitza la freqüència de cada element de la seqüència  $S$ .

A continuació cal construir l'arbre de codificació. Amb cada parell  $\langle \text{índex}, \text{freqüència} \rangle$  es construeix un arbre binari amb un sol node i s'afegeix al conjunt. A continuació es procedeix seguint els següents passos:

1. S'extreuen del conjunt els dos elements més infreqüents del conjunt i es crea un arbre binari amb un node arrel i els dos elements com a fills esquerre i dret respectivament. En el cas d'empat, si dos elements tenen la mateixa freqüència, llavors es considera més infreqüent el que té índex menor.
2. L'arbre obtingut s'afegeix com un nou element al conjunt, sent la seva freqüència la suma de les freqüències del seus dos fills i el seu índex el menor dels índexs dels seus dos fills.
3. Si encara queden dos o més elements, es torna al pas 1. Si només queda un element aquest és l'arbre binari de codificació.

La figura 3.1 mostra gràficament el procés de construcció del codi de Huffman.

L'arbre de codificació obtingut ens proporciona el codi de Huffman. Per saber el codi que li correspon a un índex només cal llegir el camí que va des de l'arrel fins la fulla que conté l'índex, llegint 0 si anem a l'esquerra i un 1 si anem cap a la dreta.

El codi de Huffman té dues propietats importants: cap codificació de cap índex és prefix de cap altra, i la seva longitud mitjana és òptima.

#### Més informació a:

- <http://www.data-compression.com>
- [http://en.wikipedia.org/wiki/Huffman\\_coding](http://en.wikipedia.org/wiki/Huffman_coding)

### 3.5.2 `arbbin.hpp`



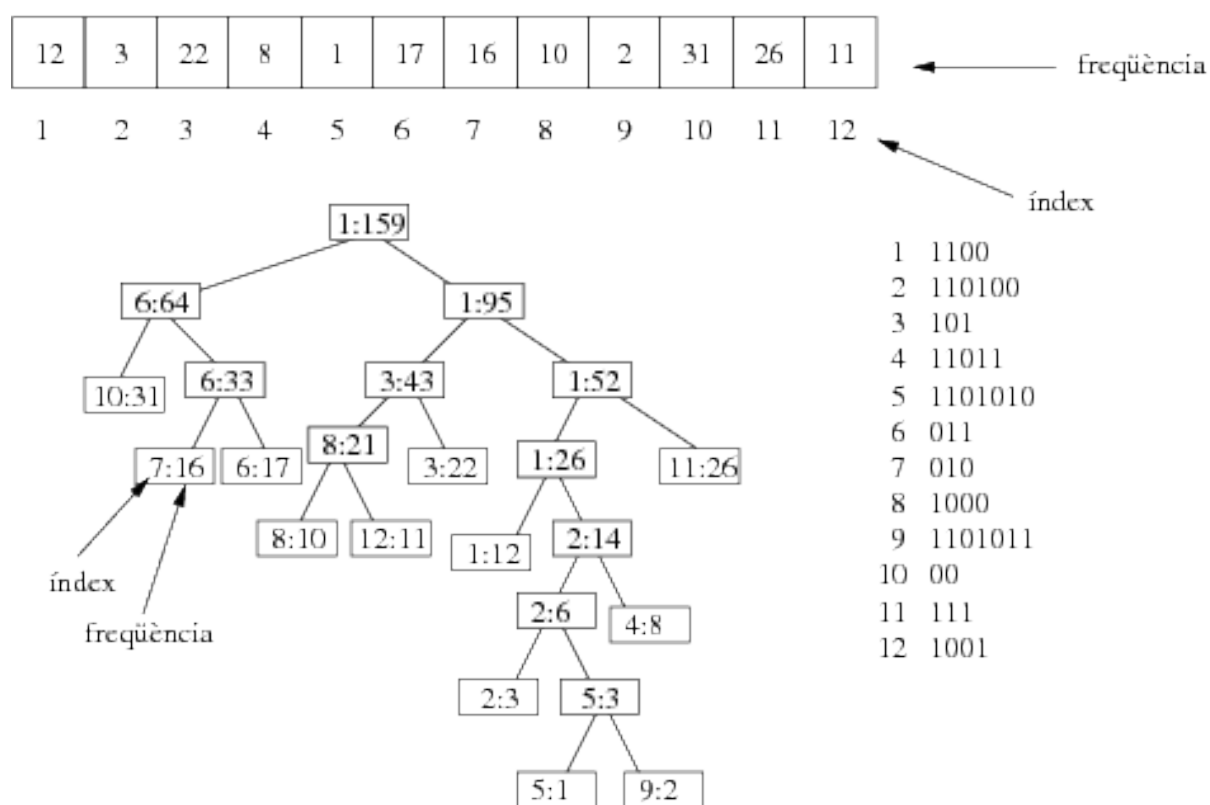


Figura 3.1: Procés per crear l'arbre de codificació de Huffman.

La classe `arbbin<Elem>` representa arbres binaris de `Elem`s. Les operacions són les habituals sobre arbres. Els elements s'emmagatzemen tant a les fulles (nodes sense fills) com en els nodes interns.

*Decisions sobre les dades:* El tipus `Elem` ha de tenir definida la constructora per defecte (`Elem()`), la constructora per còpia i l'operador assignació.

*Implementació:* La representació de les classes es trobarà en el fitxer capçalera `arbbin.hpp` (a la part privada de cada classe); la implementació d'ambdues classes estarà en el fitxer `arbbin.t`.

```
#ifndef _ARBBIN_HPP_
#define _ARBBIN_HPP_

#include <esin/error>
#include <esin/util>

using util::nat;

template <typename Elem>
class arbbin {
private:
```

Els atributs i mètodes privats de la classe `arbbin` van aquí.

```
public:
```

Un `arbbin::iterador` és un iterador sobre els elements d'un `arbbin`. Donada l'estreta relació entre les dues classes, es declara la classe `arbbin::iterador` com amiga d'`arbbin`; d'aquesta manera ambdues classes tenen accés mutu a les seves representacions privades.

```
friend class iterador;

class iterador {

    friend class arbbin;

public:
```

Crea un iterador nul.

```
iterador() throw(error);
```

Constructora per còpia, operador d'assignació i destructora.

```
iterador(const iterador& it) throw(error);
iterador& operator=(const iterador& it) throw(error);
~iterador() throw();
```

Retorna l'element del node arrel del subarbre al qual apunta l'iterador; es produeix un error si l'iterador és nul.

```
Elem operator*() const throw(error);
```

Retorna un iterador al fill esquerre de l'arbbin al qual apunta l'iterador. L'iterador que es retorna és nul si l'arbre no té fill esquerre. Es produeix un error si l'iterador és nul.

```
iterador fill_esq() const throw(error);
```

Retorna un iterador al fill dret de l'arbbin al qual apunta l'iterador. L'iterador que es retorna és nul si l'arbre no té fill dret. Es produeix un error si l'iterador és nul.

```
iterador fill_dret() const throw(error);
```

Retorna cert si i només si l'iterador és no nul.

```
operator bool() const throw();
```

Gestió d'errors i informació interna

```
static const int IteradorNul = 11;
```

```
private:
```

Els atributs i mètodes privats de la classe iterador van aquí.

```
};
```

Construeix un arbre binari buit.

```
arbbin() throw(error);
```

Construeix un arbre binari resultat de plantar els dos arbre binaris donats, com fills esquerre i dret, respectivament, d'un node arrel que conté l'element x.

```
arbbin(const Elem& x, const arbbin& fesq, const arbbin& fdret) throw(error);
```

Constructora per còpia, operador d'assignació i destructora.

```
arbbin(const arbbin& a) throw(error);
arbbin& operator=(const arbbin& a) throw(error);
~arbbin() throw();
```

Retorna el número d'elements de l'arbbin.

```
nat size() const throw();
```

Retorna un iterador a l'element que hi ha a l'arrel.

```
iterador arrel() const throw();
};
#include "arbbin.t"
#endif
```

### 3.5.3 huffman.hpp

El mòdul huffman implementa una versió de l'algorisme de huffman descrit a la subsecció 3.5.1. Per tal d'implementar aquest algorisme cal usar la classe arbbin.hpp.

*Implementació:* La implementació d'aquest mòdul es trobarà en el fitxer huffman.cpp. Es poden usar les classes vector, string i pair (fent #include <utility>) de la biblioteca estàndar de C++.

```
#ifndef _HUFFMAN_HPP
#define _HUFFMAN_HPP

#include <vector>
#include <string>

#include <esin/error>
#include <esin/util>

using util::nat;
using namespace std;

namespace huffman {
```

Donada una seqüència d'enters positius seq de longitud  $\geq 2$ , on tots els elements de seq estan en un rang  $1 \dots k$  i cadascun dels valors entre 1 i  $k$  apareix com a mínim un cop a la seqüència, aquesta funció retorna un vector taula\_codis de string's, sent taula\_codis[i] el codi (una seqüència de 0's i 1's) assignat a i seguint el procediment per generar el codi de Huffman descrit a la subsecció 3.5.1.

Es produeix un error si la seqüència és buida o només té un únic element.

```
void codi_huffman(const vector<nat>& seq, vector<string>& taula_codis) throw(error);
```

Gestió d'errors i informació interna.

```
static const int SeqCurta = 21;
```

```
};  
#endif
```





# Decàleg per implementar una classe en C++

Començar a implementar una classe en C++ des de zero 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.

## 1<sup>a</sup> 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<sup>1</sup>. 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 A.1.

## 2<sup>a</sup> 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, ...).

---

<sup>1</sup><http://atenea.upc.edu>

```

1 #ifndef _INTCELL_HPP_
2 #define _INTCELL_HPP_
3
4 #include <string>
5
6 class IntCell {
7     public:
8         explicit IntCell(int initialValue=0);
9
10        int read() const;
11        void write(int x);
12        IntCell operator+(const IntCell &ic) const;
13
14    private:
15        // Si la classe fos per una sessió de laboratori la representació s'afegiria aquí.
16
17 };
18 #endif

```

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

- ★ A l'hora de triar la representació d'una classe s'ha de pensar en l'eficiència esperada de cada una de les operacions.
- ★ 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*.

### 3<sup>a</sup> 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 A.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.



```

1 [...]
2 private:
3     // La representació de la classe IntCell està formada per un únic atribut
4     // "_storedValue", on emmagatzemem el valor de la cel·la.
5     int _storedValue;
6 };
7 #endif

```

Figura A.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 A.3 per tenir un exemple del resultat esperat d'aquest pas després d'haver transformat el fitxer capçalera.

```

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

```

Figura A.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 A.4.

```

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 A.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 A.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 A.5: Fitxer .CPP *després* del segon pas de la 5<sup>a</sup> llei

3. ESBORRAR les inicialitzacions per defecte<sup>2</sup> (=0, =1, ...) i el modificador **explicit**<sup>3</sup> de les capçaleres del .CPP. Veure la figura A.6.

```

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

```

Figura A.6: Fitxer .CPP *després* del tercer pas de la 5<sup>a</sup> llei

## 6<sup>a</sup> 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 A.7.

```

1 #include "intcell.hpp"
2
3 IntCell::IntCell(int initialValue) { }
4 int IntCell::read() const { }
5 void IntCell::write(int x) { }
6 IntCell IntCell::operator+(const IntCell &ic) const { }

```

Figura A.7: Fitxer .CPP *després* de la 6<sup>a</sup> llei

<sup>2</sup>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.

<sup>3</sup> 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.

## 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 funcionament. 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 A.8.

```

1 #include <iostream>
2 #include "intcell.hpp"
3
4 using namespace std;
5
6 int main () {
7     IntCell icell;
8     cout << icell.read() << endl;
9 }
```

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

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
```

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 A.9.

```
1 #include "intcell.hpp"
2
3 IntCell::IntCell(int initialValue) {
4     _storedValue = initialValue;
5 }
6
7 int IntCell::read() const {
8     return _storedValue;
9 }
10
11 void IntCell::write(int x) {
12     _storedValue = x;
13 }
14
15 IntCell IntCell::operator+(const IntCell &ic) const {
16     IntCell nou(_storedValue + ic._storedValue);
17     return nou;
18 }
```

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

# B

## Compilació, muntatge i execució en C++

Un compilador és un programa d'ordinador que permet traduir un programa escrit (un llenguatge d'alt nivell) a un altre llenguatge de programació (normalment llenguatge màquina), generant un programa equivalent que l'ordinador pot entendre.

Aquesta eina permet al programador desconèixer el llenguatge que utilitza l'ordinador i escriure en un llenguatge més universal i més proper a com pensa un ésser humà.

En aquest capítol veure'm com és el procés de compilació, muntatge i execució en C++, i diferents eines que ens poden ser útils en cadascuna d'aquestes etapes.

### B.1 Compilació separada i muntatge

#### B.1.1 Compilació i muntatge bàsic

La compilació és el procés durant el qual es tradueixen les instruccions escrites en un determinat llenguatge de programació a llenguatge màquina.

Per compilar programes en C++ utilitzarem el compilador GNU `gcc`, en concret utilitzarem l'ordre `g++`.

Per compilar un fitxer font per separat s'usa l'ordre `g++` (el text en cursiva indica un argument) amb l' **opció -c** :

```
$ g++ -c nom_fitxer.cpp
```

Aquesta ordre produeix un fitxer objecte *nom\_fitxer.o*.

Per generar el fitxer executable cal muntar ("linkar") un o més fitxers objectes. Només cal posar els noms dels fitxers objecte (sense que importi l'ordre) darrera de `g++`:

```
$ g++ nom_fitxer1.o nom_fitxer2.o ...
```

Aquest procés genera el fitxer executable per defecte que s'anomena `a.out`. Si es vol que el fitxer executable tingui un nom diferent llavors s'ha d'usar l' **opció -o** :

```
$ g++ -o nom_executable.e nom_fitxer1.o nom_fitxer2.o ...
```

També es pot compilar i muntar varis fitxers en una única ordre:

```
$ g++ -o nom_executable.e f1.cpp f2.o ...
```

Per exemple, donades dues classes (*a* i *b*) que són utilitzades per un programa principal (*prog*) (veure la representació en la figura B.1) es podria generar l'executable d'aquesta forma:

```
$ g++ -o prog.e a.o prog.cpp b.o
```

Es genera un fitxer executable anomenat *prog.e*, que es forma a partir de la compilació del fitxer *pr.cpp* i el fitxer objecte resultant es munta amb els fitxers *a.o* i *ba.o*. El fitxer intermig *pr.o* no es conserva.

## B.1.2 Compilació i muntatge de classes genèriques

Els mòduls que defineixen classes o funcions genèriques no es compilen MAI per separat. Una forma adequada d'organitzar el codi consisteix en escriure la classe en dos fitxers:

- *declaració de la classe*: en el fitxer capçalera amb extensió *.hpp* com estem acostumats.
- *implementació de la classe*: en un fitxer que per conveni li donarem l'extensió *.t*.

Cal tenir en compte que el fitxer *.hpp* ha d'incloure al fitxer *.t*. Així doncs, si un programa usa a la classe genèrica *X* llavors haurà d'incloure el fitxer *X.hpp* (i indirectament inclourà a *X.t*).

Es pot obtenir una comprovació sintàctica de la classe genèrica mitjançant la inclusió del fitxer *.hpp* (i per tant del fitxer *.t*) en un fitxer *.cpp*. Aquest fitxer només cal que tingui la línia d'inclusió.

Per exemple, donades dues classes *a* i *b*, que són utilitzades per un programa principal (*prog*) i la classe *a* és genèrica (veure la representació en la figura B.2) es podria generar l'executable després de fer:

```
$ g++ -c prog.cpp
$ g++ -c b.cpp
$ g++ -o prog.e prog.o b.o
```

Com es pot comprovar la classe *a* no s'hauria de compilar.

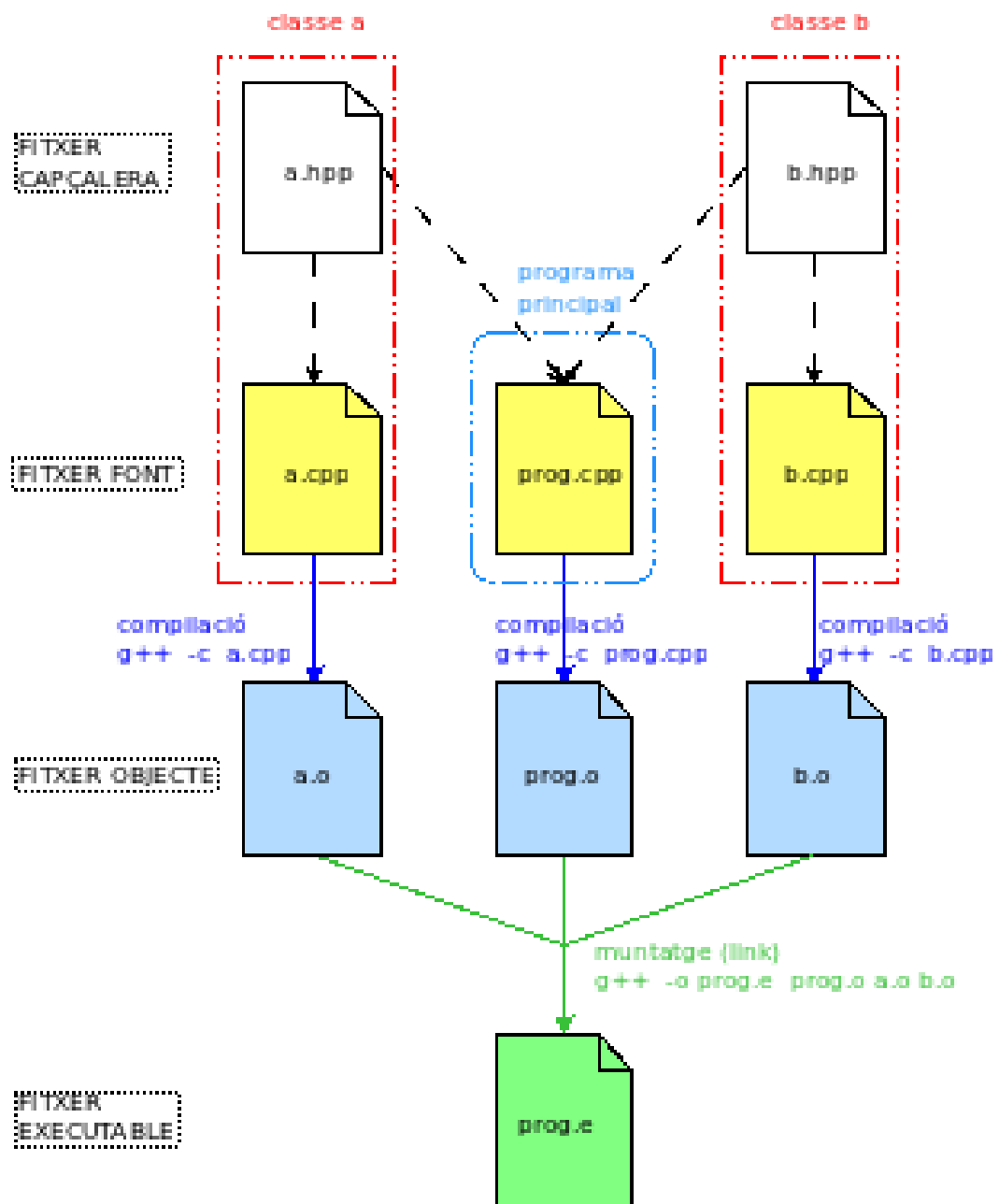


Figura B.1: Compilació i muntatge de les classes *a* i *b*, i el programa principal *prog*.

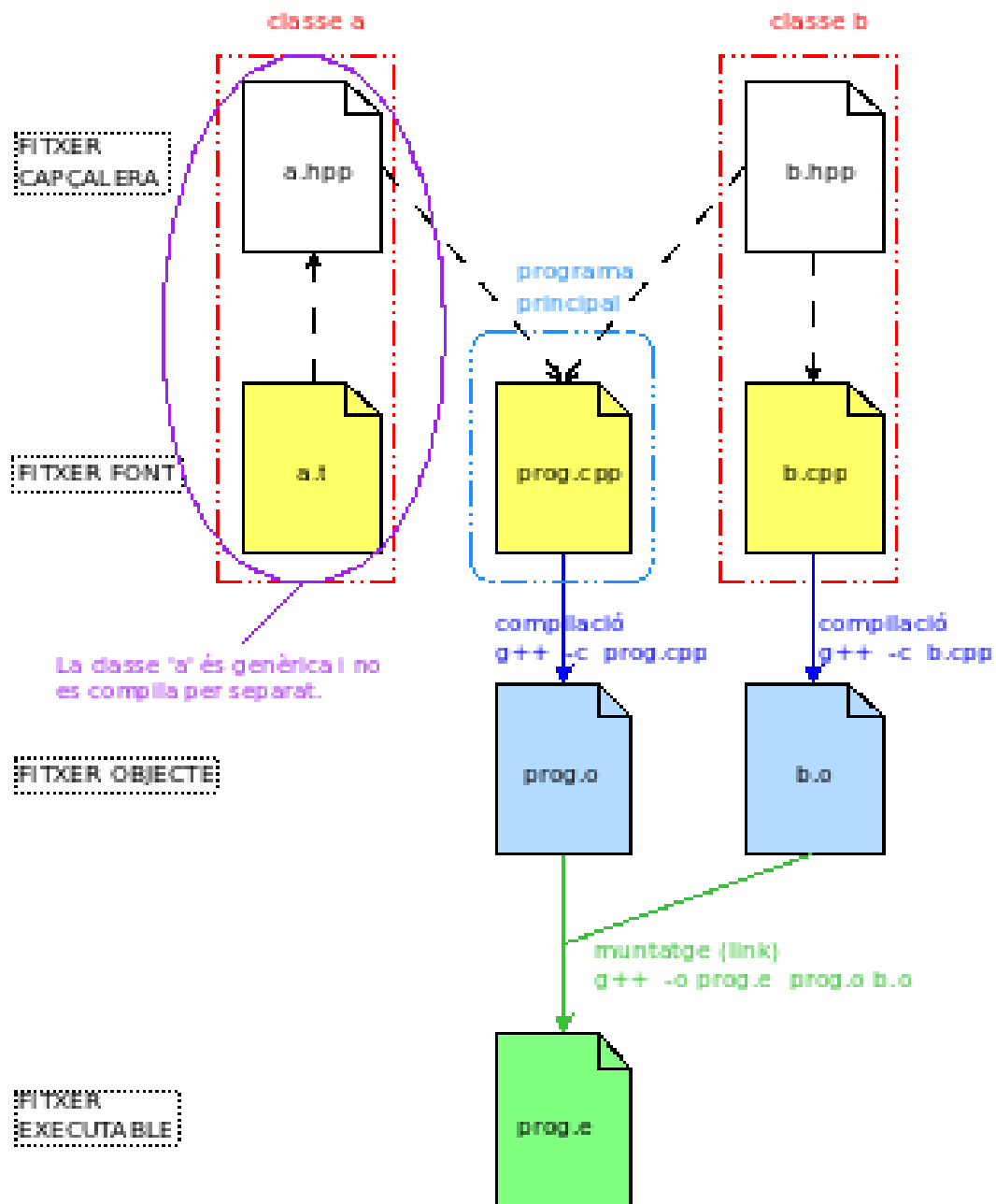


Figura B.2: Compilació i muntatge de les classes *a* (genèrica) i *b*, i el programa principal *prog*.



### B.1.3 Compilació i muntatge amb biblioteques

Molts programes només utilitzen classes, mètodes i funcions definides a la biblioteca estàndard (per exemple, `string`, `iostream`, `list`, ...). El muntador (o *linker*) sempre empra per defecte aquesta biblioteca.

Si volem emprar una altra biblioteca caldrà indicar-ho explícitament en l'ordre de muntatge mitjançant l' **opció -l**.

A Unix el conveni és anomenar a les biblioteques `libxxxx` amb extensió `.a` o `.so` (depenent de si la biblioteca és estàtica o dinàmica. Després de l'opció `-l` es posa la part variable del nom de la biblioteca, és a dir, `xxxx`).

Per exemple, per usar les biblioteques `libB1` i `libB2` caldria executar la següent ordre:

```
$ g++ -o nom_executable.e f1.o f2.o ... -lB1 -lB2
```

### B.1.4 Altres opcions

#### Opció -I

Si necessitem utilitzar fitxers de capçalera (`.hpp`) que no es troben en el mateix directori on estem compilant o en un directori estàndard d'inclusió (per exemple, `/usr/include`) caldrà utilitzar l' **opció -I**. Posarem tantes opcions `-I` com directoris volguem afegir a la llista de directoris d'inclusió.

```
$ g++ -c -I /home/users/adam/headers f1.cpp
```

Per especificar un camí amb l'opció `-I` es pot usar el camí absolut o el camí relatiu.

Si l'ordre de l'exemple anterior s'estigués executant en el directori `/home/users/eva/pract` podríem haver escrit:

```
$ g++ -c -I ../../adam/headers f1.cpp
```

#### Opció -L

Un problema similar a l'anterior es dona si necessitem usar una biblioteca que no estigui en el directori en curs o en un directori estàndard (per exemple, `/usr/lib`). En aquest cas s'utilitza l' **opció -L** per indicar el camí.

Per exemple, suposem que volem utilitzar `libB1.a` que es troba a `/home/users/esin`. Llavors escriurem:

```
$ g++ -L /home/users/esin -o prog.e prog.cpp f1.cpp -lB1
```

#### Opció -Wall

Per aconseguir que el compilador verifiqui instruccions dubtoses i avisi del màxim nombre possible de fonts d'error cal utilitzar l' **opció -Wall**:

```
$ g++ -c -Wall nom_fitxer.cpp
```

## Opció -g

Si es vol utilitzar un *debugger* caldrà afegir l' `opció -g` al compilar:

```
$ g++ -g -c -Wall fitx.cpp
```

o si es crea directament l'executable (sense compilació separada) llavors:

```
$ g++ -g -Wall -o nom_executable.e f1.cpp f2.cpp ...
```

## B.2 Make

make és un programa d'Unix que simplifica notablement el treball de compilació i muntatge. A més a més, es pot instruir adequadament a make per tal que recompili només aquells fitxers que facin falta.

El programa make utilitza un fitxer anomenat Makefile que s'ha de trobar en el mateix lloc on s'executa el make. Aquest programa té un argument, el *target*, que és el que volem que es construeixi. Si escrivim make sense cap argument, el *target* és el primer que aparegui en el Makefile.

Per veure el funcionament del make escriurem el Makefile per l'exemple de la figura B.1. Per complicar-ho una mica suposarem que els fitxers .hpp es troben en el directori: /home/users/yo/elsmeusincludes, i a més a més necessitem la biblioteca /home/users/yo/elsmeuslibs/libB1.a.

```
1 CC = g++
2 INCL = /home/users/yo/elsmeusincludes
3 COMPILE = $(CC) -c -Wall -I $(INCL)
4 LIBS = /home/users/yo/elsmeuslibs
5 LINK = $(CC) -L $(LIBS)
6 OBJS = prog.o a.o b.o
7
8 prog.e: $(OBJS) $(LIBS)/libB1.a
9         $(LINK) -o prog.e $(OBJS) -lB1
10 prog.o: prog.cpp $(INCL)/a.hpp $(INCL)/b.hpp
11         $(COMPILE) prog.cpp
12 a.o: a.cpp $(INCL)/a.hpp
13         $(COMPILE) a.cpp
14 b.o: b.cpp $(INCL)/b.hpp
15         $(COMPILE) b.cpp
```

Les primeres sis línies defineixen variables. Es poden declarar tantes variables com vulguem. Si X és una variable, es pot accedir al seu valor mitjançant \$(X). Després de la declaració de variables vénen una sèrie de blocs de la forma:

```
target: dependències
        ordre1
        ordre2
        ...
```

Cada línia en la que escrivim una ordre ha de començar obligatòriament amb un **tabulador**.

Les dependències són llistes que indiquen quins elements intervenen directament en la construcció del *target* i poden canviar.

Per exemple, *prog.e* depèn dels fitxers *prog.o*, *a.o*, *b.o* i també de *libB1.a*, per això apareixen aquests fitxers en la llista de dependències. Sota la llista apareix l'ordre per generar *prog.e* a partir de les dependències. I el mateix passa amb la resta.

Es poden fer moltes altres coses mitjançant *make*. No només facilita el procés de compilació i muntatge.

Per exemple, si afegim aquestes línies al final del *Makefile*:

```
1 clean:
2      rm -rf *.o; rm prog.e
```

cada cop que escrivim *make clean* esborrarem tots el fitxers *.o* i l'executable.

## B.3 Execució

Per executar un programa només cal escriure el nom de l'executable després del *prompt* de Unix:

```
$ nom_executable
```

Si el programa llegeix i escriu pels canals estàndard d'entrada (*cin*) i sortida (*cout*) es pot redirigir l'entrada per tal que vingui d'un fitxer i no del teclat, i la sortida per tal que s'escrigui en un fitxer enlloc de la pantalla. Per fer això utilitzarem els operadors de redireccionament (*<* per l'entrada i *>* per la sortida) seguits del nom del fitxer corresponent. Es pot redirigir només l'entrada, només la sortida, o ambdues:

```
$ nom_executable < fitxer_entrada > fitxer_sortida
```

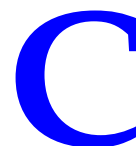
Es pot connectar la sortida estàndard d'un programa amb l'entrada estàndard d'un altre programa mitjançant una *pipe*. L'operador corresponent és la barra vertical *|*.

```
$ nom_executable1 | nom_executable2
```

Molts programes escriuen els seus missatges d'error pel canal estàndard d'error (*cerr*) que normalment està associat a la pantalla. Si utilitzem l'operador *>* els missatges continuen apareixent per pantalla. Per redirigir el canal *cerr* s'utilitza l'operador *>&* seguit del nom del fitxer. Per exemple:

```
$ g++ -c pr.cpp >& errores.txt
```





# Estil de programació i documentació

Un estil de programació consistent i una documentació clara, concisa i precisa són dos elements claus d'un bon programa. La detecció d'errors o la modificació d'un programa mal documentat i sense una mínima coherència (noms dels identificadors, sagnat<sup>1</sup>, estructuració dels components del programa, etc.) és una tasca gairebé impossible. A continuació us donem una sèrie de recomenacions sobre aquests dos aspectes.

## C.1 Noms de variables adequats

**Useu noms descriptius per constants, classes i funcions visibles en diferents punts del programa, i noms breus per les variables locals o els paràmetres formals d'una funció.**

Per exemple, és inapropiat cridar a un mètode *f* o a una classe *X* (menys en el cas que es vulgui il·lustrar una característica del llenguatge de programació!). Els identificadors pels mètodes o les classes han de descriure el seu propòsit i per tant solen ser llargs i estar compostos per vàries paraules: *copia\_pila*, *insereix*, *SopaLletres*, ... En canvi, per un paràmetre formal o una variable local l'identificador *n* és adequat, *npunts* acceptable i *numero\_de\_punts* és un usar un "canó per matar mosques".

Si una variable local s'utilitzarà d'una manera convencional podem donar-li un identificador mínim: *i*, *j* i *k* s'acostumen a usar pels índexos de bucles, *p* i *q* per punters, *s* i *t* per *strings*, etc. Per exemple:

```
1 void inicia_taula(elem taula_elems[], int nr_elems) {  
2     int indice_elem;  
3     for (indice_elem = 0; indice_elem < nr_elems;  
4                                     indice_elem++) {
```

---

<sup>1</sup>"Indentació".

```

5     taula_elems[indice_elem] = indice_elem;
6     }
7 }

```

és un codi molt menys entenedor que

```

1 void inicia_taula(elem a[], int n) {
2     for (int i = 0; i < n; i++) {
3         a[i] = i;
4     }
5 }

```

## C.2 Conveni consistent pels identificadors

**“Inventeu” un esquema per expressar els identificadors i apliqueu-lo sistemàticament i consistent.**

Per exemple, s’acostuma a recomenar utilitzar noms en majúscules per constants (p.e., *ELEM\_SIZE*, *MAX\_ELEMS*). Un conveni que hem utilitzat en aquest document i en el material de l’assignatura és el de combinar majúscules i minúscules pels identificadors de codis i missatges d’error (p.e., *NoSolReal*, *PilaPlena*). En general, les connectives (p.e., articles i preposicions) es deixen en minúscules. Alguns programadors utilitzen el conveni descrit per tots els seus identificadors de funcions, classes, i altres elements globals: *class Pila* { ... }, *CopiaPila*, *elCim*, *unNode*, ... Altres prefereixen utilitzar minúscules i separar les paraules mitjançant el símbol de subratllat: *copia\_pila*, *el\_cim*, ... Un conveni que també té molts adeptes és el d’anteposar el caràcter de subratllat als elements privats d’una classe:

```

1 template <typename T>
2 class pila {
3     ...
4     private:
5         int _cim;
6         T _cont;
7         int _max_elems;
8 };

```

## C.3 Utilitzar noms en forma activa per les funcions

**Es recomana usar verbs en veu activa pels mètodes i les funcions, i reservar adjectius o noms de la forma *es...* per funcions o mètodes el resultat dels quals és un booleà.**

Convé pensar els identificadors des del punt de vista de l'usuari, no de l'implementador i tenir en compte que els mètodes s'apliquen sobre objectes. Per exemple, compareu:

```

1 conjunt c;
2 if (c.pertany(x))
3 ...

```

amb

```

1 conjunt c;
2 if (c.conte(x))
3 ...

```

El conveni o esquema de noms utilitzat ha de basar-se en el que les entitats representen, no a com ho representen. Són desaconsellables per tant els convenis basats en una característica de baix nivell o lligada a la implementació, com per exemple anteposar el prefix *i\_* a les variables i atributs de tipus enter i el prefix *f\_* a les variables i atributs de tipus real (`float`).

Sobretot, és important ser consistent: al principi pot alentir una mica el treball haver de recordar els convenis que hem triat; però després ho farem quasi sense pensar.

Acabem aquest punt amb un exemple d'inconsistència caricaturitzat, però indicatiu de la importància que té aquest aspecte de l'estil:

```

1 class Pila {
2 public:
3     Pila(int max_elements);
4     ~Pila();
5     Pila(const Pila& s);
6     Pila& operator=(const Pila& la_pila);
7
8     void Apilar(int element);
9     int desapila();
10    int Cim();
11    bool Buida();
12
13 private:
14     struct tnode {
15         int info;
16         tnode* seguent;
17     };
18     tnode* cim;
19     int Nr_Elems_Pila;
20     int maxElems;
21 };

```

## C.4 Nom d'identificadors precisos

**Un nom no només identifica; comporta informació. Un nom inadequat induirà a confusions i errors.**

Per exemple, la implementació següent no és gens adequada:

```

1 bool conjunt::conte(int x) {
2     bool trobat = true;
3     node* p = primer;
4
5     while (p != NULL and trobat) {
6         trobat = p -> info != x;
7         p = p -> seg;
8     }
9     return trobat;
10 }
```

la variable local *trobat* significa el contrari del que el seu nom indica, i la funció retorna el contrari del que el seu nom indica. Probablement es tracta d'un error induït per l'identificador *trobat*, però canviar `return trobat` per `return not trobat` no és una bona solució.

## C.5 Identació del codi

**Sagneu el codi i utilitzeu els parèntesis i espais en blanc per millorar la llegibilitat del codi.**

La majoria d'editors de text moderns (p.e., EMACS) incorporen identació automàtica, de manera que no porta massa problema. També convé substituir, al final, tots els tabuladors (introduïts mitjançant la tecla TAB o automàticament per l'editor) per espais en blanc. Sinó les impressions en paper poden quedar desajustades. Useu els parèntesis i els espais en blanc per resoldre ambigüetats i facilitar la comprensió de les expressions. Per exemple,

```

1 bool es_any_traspas(int y) {
2     return y%4==0 and y%100!=0 or y%400==0;
3 }
4 ...
5 for(int i=0;i<n;i++)
6     atraspas[i]=es_any_traspas(llista_anys[i]);
7 ...
```



és, per la majoria de la gent, més difícil de comprendre que

```

1 bool es_any_traspas(int y) {
2     return ((y % 4 == 0) and (y % 100 != 0)) or
3           (y % 400 == 0);
4 }
5 ...
6 for (int i = 0; i < n; ++i) {
7     atraspas[i] = es_any_traspas(llista_anys[i]);
8 }
9 ...

```

Això mateix passa amb les claus ({} ) d'obertura i tancament de blocs. Si un bloc només conté una instrucció no fa falta usar-les, però pot ser útil per millorar la llegibilitat i evitar errors com en el següent exemple:

```

1 if (mes == FEBRER) {
2     correcte = true;
3     if (es_any_traspas(any))
4         if (dia > 29)
5             correcte = false;
6     else // aquest else NO emparella amb el segon if!!
7         if (dia > 28)
8             correcte = false;
9 }

```

Es pot resoldre el problema usant les claus en el lloc adequat o millor encara escrivint:

```

1 if (mes == FEBRER) {
2     correcte = (dia <= 28) or
3               (dia == 29 and es_any_traspas(any));
4 }

```

Sigueu consistents amb l'estil d'identificació i l'ús de les claus. Alguns estils de sagnat i ús de les claus populars són:

1. les claus s'obren i es tanquen en línies separades;

```

// Exemple de l'estil 1
for (int j = 0; j < k; j++)
{
    i = j % 2;
    if (i == 0)
    {
        ...
    }
    else

```

```

    {
        ...
    }
}

```

2. la clau s'obre en la línia que obre el bloc `for`, `while`, etc., i les claus de tancament van en línies separades;

```

// Exemple de l'estil 2
for (int j = 0; j < k; j++) {
    i = j % 2;
    if (i == 0) {
        ...
    }
    else {
        ...
    }
}

```

3. totes les claus de tancament consecutives es posen en la mateixa línia.

```

// Exemple de l'estil 3
for (int j = 0; j < k; j++) {
    i = j % 2;
    if (i == 0) {
        ...
    } else {
        ...
    }
}

```

Convé usar algun dels estils usals ja que són ben soportats pels editors de text i no resultaran xocants per altra gent que llegeixi el codi.

Una excepció respecte les regles habituals de sagnat són les alternatives múltiples. En C i C++ és típic escriure:

```

if (B1)
    S1
else if (B2)
    S2
...
else if (Bn)
    Sn
else
    Sn+1

```

amb tots els `else`'s aliniats; quan les  $B_i$ 's són mutuament excluyents (els `if ... else`'s s'avaluen seqüencialment i no hi ha indeterminisme).

## C.6 Evitar una lògica del programa antinatural

Eviteu un “flux” o lògica del programa antinatural i “factoritzeu” el codi comú.

Per exemple, en lloc de

```

1 int s = 0;
2 node* p = _primer;
3 if (p == NULL) { // la llista és buida; no fem res
4 }
5 else {
6     while (p != NULL) {
7         s = s + p -> valor;
8         p = p -> seg;
9     }
10 }
11 return s;
```

hauríem d'haver escrit:

```

1 int s = 0;
2 node* p = _primer;
3 while (p != NULL) {
4     s += p -> valor;
5     p = p -> seg;
6 }
7 return s;
```

o bé

```

1 int s = 0;
2 for (node* p = _primer; p != NULL; p = p -> seg) {
3     s += p -> valor;
4 }
5 return s;
```

Un altre exemple és la següent funció per insertar un element en un conjunt implementat mitjançant una llista enllaçada ordenada, amb fantasma i apuntadors al primer i a l'últim node:

```

1 void conjunt::insereix(const string& x) {
2     node* q = _primer;           // q apunta al fantasma
3     while (q -> seg != NULL and q -> seg -> info < x) {
4         q = q -> seg;
5     }
6     if (q -> seg == NULL) {
7         node* p = new node;      // el nou node serà l'últim
8         p -> info = x;
9         p -> seg = NULL;
10        _ultim = p;
11        q -> seg = p;
12    }
13    else if (q -> seg -> info == x) { // no es fa res
14    }
15    else {
16        node* p = new node;
17        p -> info = x;
18        p -> seg = q -> seg;
19        q -> seg = p;
20    }
21 }

```

Hauria estat molt millor “factoritzar” la part comuna i simplificar la “lògica” de la part que segueix al bucle de cerca:

```

1 void conjunt::insereix(const string& x) {
2     node* q = _primer; // q apunta al fantasma
3     while (q -> seg != NULL and q -> seg -> info < x) {
4         q = q -> seg;
5     }
6     if (q -> seg == NULL or q -> seg -> info != x) {
7         node* p = new node;
8         p -> info = x;
9         p -> seg = q -> seg;
10        if (q -> seg == NULL) {
11            _ultim = p;
12        }
13        q -> seg = p;
14    }
15 }

```

## C.7 Disminuir la complexitat

**Disminuïu la complexitat del vostre codi mitjançant un ús assenyat de la descomposició funcional. Aproveiteu les solucions a problemes similars mitjançant una adequada descomposició funcional.**

Per exemple, en una classe implementada amb una llista enllaçada és freqüent tenir un bucle, com a l'exemple previ, o el codi corresponent a la inserció en un punt concret de la llista. Per tant pot ser convenient tenir sengles operacions privades i implementar les operacions públiques usant les privades:

```

1 // llista_ord.hpp
2 class llista_ord {
3     public:
4         ...
5         // insereix 'x' a la llista, en ordre
6         void insereix(int x);
7         ...
8     private:
9         ...
10        static void insereix_darrere(node* p, int x);
11        node* troba_elem(int x);
12
13 };
14
15 // llista_ord.cpp
16 ...
17 void llista_ord::insereix(int x) {
18     node* q = troba_elem(x);
19     if (q -> seg == NULL or q -> seg -> info > x) {
20         insereix_darrere(q, x);
21     }
22 }
23
24 // mètode privat
25 // retorna un apuntador a l'últim node de la llista tal que la seva info és <
26 // x. En alguns casos retornarà el fantasma, si la llista és buida o el primer
27 // de la llista és >= x, o l'últim node, si x és major que la info de qualsevol
28 // node de la llista.
29 llista_ord::node* llista_ord::troba_elem(int x) {
30     node* q = _primer;
31     while (q -> seg != NULL and q -> seg -> info < x) {
32         q = q -> seg;
33     }

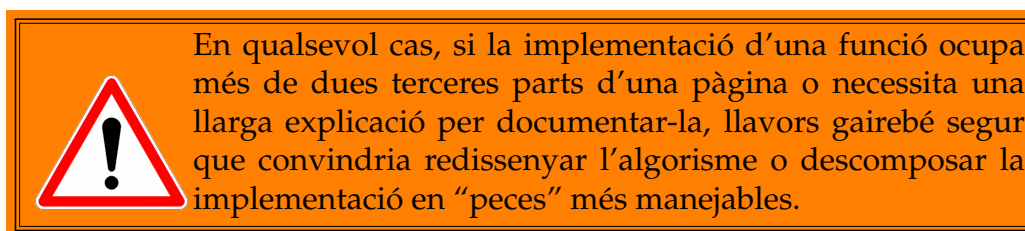
```

```

34     return q;
35 }
36
37 // mètode privat de classe
38 // insereix a la llista enllaçada un nou node, amb info == x, com a successor
39 // del node apuntat per p. Pre: p != NULL
40 void llista_ord::insereix_darrere(node* p, int x) {
41     node* n = new node;
42     n->info = x;
43     n->sig = p->sig;
44     p->sig = n;
45 }
46 ...

```

Donat que només els mètodes de la classe poden utilitzar als mètodes privats és adequat suposar que les precondicions es compliran en ser invocats, i així evitem una gestió d'errors complexa.



Un altre aspecte a considerar és l'ordre en el que definim les funcions. Existeixen diverses alternatives raonables: totes les funcions privades en primer lloc i després les públiques; o al revés. Un conveni probablement millor és el de situar les operacions privades el més properes (just abans o just després) de l'operació o operacions que les usin.

## C.8 Useu construccions similars per tasques similars

**Useu construccions similars per realitzar tasques similars.**

Si en un punt del programa iniciu una taula  $t$  mitjançant:

```

1 for (int i = 0; i < n; ++i)
2     t[i] = 0;

```

llavors no escriviu el bucle que calcula quants elements no nuls hi ha en  $t$  de la següent manera encara que sigui totalment correcte:

```

1 i = 0;
2 nnuls = 0;
3 while (i <= n - 1) {
4     if (A[i] != 0) {
5         ++nnuls;
6     }
7     ++i;
8 }

```

## C.9 No usar variables globals

No useu variables globals, exceptuant quan sigui estrictament imprescindible. Una variable o objecte global és extern a qualsevol funció o mètode. Els atributs de classe són bàsicament objectes globals, exceptuant que l'accés a ells pot restringir-se si es declaren a la part privada.

```

int nr_elems;           // variable global! MALAMENT
const int MAX_ELEMS = 30; // constant global, OK

class X {
    ...
    static const int MAX_SIZE = 20; // constant de classe, OK
    static int nr_objectes;         // variable de classe! MALAMENT
    ...
};

```

El problema amb els objectes globals és que podem tenir efectes laterals en les funcions i mètodes, i es trenquen els principis de modularitat. En el següent exemple la funció *esta* només funciona per l'array *t* la mida del qual és *n*, i no obstant això, l'algorisme que implementa és igualment vàlid per qualsevol array:

```

1 // variables globals
2 int t[20];
3 int n;
4
5 // retorna cert si i només si 'x' està en t[0..n-1]
6 bool esta(int x) {
7     for (int i = 0; i < n and t[i] != x; ++i)
8         ; // el cos del bucle és buit
9     return i < n;
10 }

```

Tota comunicació entre les funcions i els mètodes amb el seu entorn hauria de produir-se a través dels seus paràmetres o retorns. Observeu que per un mètode d'una classe *X* l'objecte al qual s'aplica el mètode és un paràmetre implícit i per tant no suposa una violació d'aquesta regla.

Les anomenades *variables locals estàtiques* són una altra forma encoberta de trencar la modularitat. Una variable d'aquest tipus és una variable local a una funció o mètode, però reté el seu valor entre execucions successives.

Hi ha casos excepcionals i plenament justificats per l'ús de variables globals o estàtiques; p.e., els objectes `cout`, `cin` i `cerr` són objectes globals. Fixeu-vos, no obstant, que acostumem a definir funcions del tipus `print` o els operadors `<<` i `>>` de manera que reben un paràmetre de tipus *ostream* o *istream* explícit.

Un exemple clàssic d'ús justificat de variables estàtiques i globals és un generador de nombres pseudo-aleatoris: cada número és generat a partir de l'anterior (exceptuant la primera vegada) i no és adequat ni còmode que l'usuari haig de gestionar-ho a través de paràmetres explícits:

```

1 double llavor = 0.0; // variable global
2 void inicialitza_rand(double sm) {
3     llavor = sm;
4 }
5 double rand() {
6     static double x = llavor;
7     // la primera vegada s'inicialitza amb el valor de la variable global;
8     // i a partir d'aquest moment, la variable estàtica local x comença amb
9     // el seu valor en l'execució prèvia
10    x = funcio_complicada(x);
11    return x;
12 }
```

El llenguatge C++ ens ofereix mecanismes que ens permeten donar una solució més “neta” a aquest tipus de situacions (només per mencionar un defecte de l'exemple anterior, observeu que res impediria que qualsevol funció accedís i modifiqués la variable *llavor*).

En particular podem usar variables privades de classe :

```

1 class Random {
2     public:
3         Random(double ll = 0.0) { _x = ll; };
4         double rand() { _x = funcio_complicada(_x); return _x; };
5     private:
6         static double _x; // variable de classe
7 }
```

Una altra excepció a la regla són les variables globals que en realitat s'usen com constants globals, però que no són declarades com `const` perquè:

- no poden ser inicialitzades en un únic pas, o



- el seu valor inicial ha de ser calculat algorísmicament, o
- existeix la necessitat de poder efectuar (molt ocasionalment) canvis en el seu valor.

És usual que aquestes variables també s'implementin usant variables de classe privades, per restringir la seva manipulació i impedir en la mesura del que sigui possible usos incorrectes.

## C.10 Utilitzar variables locals

**Utilitzeu variables locals i eviteu mètodes o funcions amb llargues llistes de paràmetres. No incloeu atributs en un objecte o paràmetres en una operació innecessaris si la seva missió es pot realitzar mitjançant una o més variables locals.**

Per exemple, si una classe llista no necessita la noció de punt d'interès llavors és absurd incloure aquest tipus d'atribut per fer un recorregut o posar un punter com paràmetre d'una funció que fa el recorregut iterativament:

```

1 // usar var. local, NO un atribut _actual!
2 bool llista::conte(const T& elem) const {
3     _actual = _primer;
4     while (_actual != NULL and _actual -> info < x) {
5         _actual = _actual -> sig;
6     }
7     return _actual != NULL and _actual -> info == x;
8 }
9
10 // usar var. local, NO un paràmetre 'p'!
11 bool llista::conte_priv(const T& elem, node* p) {
12     while (p != NULL and p -> info < x) {
13         p = p -> sig;
14     }
15     return p != NULL and p -> info == x;
16 }
17
18 // OK; aquí 'p' no és un punter pel recorregut, en realitat
19 // representa a la subllista que queda por explorar
20 bool llista::conte_rec(const T& elem, node* p) {
21     if (p == NULL) {
22         return false;
23     }
24     if (p -> info >= x)
25         return p -> info == x;
26 }
27 return conte_rec(x, p -> sig);

```

28 }

## C.11 Codi ben estructurat

El codi ha de ser estructurat: cada bloc ha de tenir un únic punt d'entrada i un únic punt de sortida. No useu `breaks` (només en els `switchs`), `continues` o `gotos`. No feu `return` dins de l'interior d'un bucle. Useu l'esquema de cerca quan sigui adequat, no un `return` o `break` des de l'interior d'un bucle que fa un recorregut.

Tampoc és un bon estil “trençar” la iteració modificant la variable de control que recorre la seqüència:

```

1 for (i = 0; i < n; ++i) {
2     if (A[i] == x) {
3         i = n;      // Això és un nyap
4     }
5     ...
6 }
```

Les úniques desviacions acceptables respecte a aquesta regla són:

- les excepcions —que, per definició, trenquen immediatament el flux normal d'execució
- els `switchs`, i
- les composicions alternatives (no internes a un bucle) típiques de funcions recursives:

```

1 if (i > 1) {
2     result = x;
3 }
4 else if (i == 1) {
5     result = y;
6 }
7 else { // if(i; 1)
8     result = z;
9 }
10 return result;
```

es pot escriure

```

1 if (i > 1) {
2     return x;
3 }
4 else if (i == 1) {
```

```

5     return y;
6 }
7 else {    // if(i;1)
8     return z;
9 }

```

o fins i tot

```

1 if (i > 1) {
2     return x;
3 }
4 if (i == 1) {
5     return y;
6 }
7 return z;

```

## C.12 Bona documentació

**Una bona documentació és essencial. La representació d'una classe ha de contenir una explicació detallada de com la implementació concreta representa els valors abstractes i com és aquesta representació (invariant).**

Per exemple,

```

1 class llista {
2     ...
3     private:
4         struct node {
5             string clau;
6             int valor;
7             node* seg;
8         };
9         ...
10        nodo* _primer;
11 };

```

no ens diu si la llista està implementada mitjançant una llista simplement enllaçada, si està tancada circularment o no, si hi ha o no un node fantasma, si està ordenada o no creixentment pel camp `clau` de cada node, etc. Per tant, s'ha de documentar adequadament la forma en que la representació serà utilitzada.

No comenteu codi autoexplicatiu ni repetiu el que és obvi. Aquí teniu uns quants exemples de comentaris inútils i superflus:

```
1 // retornem cert si trobat és cert
2 return trobat;
3
4 // incrementem el comptador
5 cont++;
6
7 // inicialitzem a 0 totes les components de 'v'
8 for (int i = 0; i < n; i++)
9     v[i] = 0;
```

Un comentari ha d'aportar informació que no és immediatament evident. Generalment, convé efectuar un comentari general previ sobre el comportament de cada funció o mètode, amb indicacions sobre els punts subtils de la implementació. Eviteu l'ús de comentaris intercalats amb el propi codi, llevat que resultin absolutament imprescindibles. La documentació no és un substitut adequat d'una descomposició funcional correcta, de manera que si la implementació d'un determinat mètode o funció és llarga i complexa, la solució no és posar abundants comentaris sino descomposar-la en "peces" manejables i autoexplicatives (vegeu els exemples de les seccions C.6 i C.7).

# Índex alfabètic

## A

aliasing, 12  
arrays, *Vegeu* taules

## C

constructor per còpia, 14

## D

dangling references, 11  
`delete`, 9  
`delete[]`, 9  
destructor, 15

## E

emparellament incorrecte, 11  
estil d'identació, 87  
`explicit`, 72

## I

identació del codi, 86  
inicialitzacions per defecte, 72

## L

list, 56  
iteradors, 57

## M

memory leaks, 11  
modularitat, 93

## N

`new`, 9  
`new[]`, 9  
NULL, 2  
delete de NULL, 13  
desreferència de NULL, 12

## O

`operator*`, 1  
`operator=`, 16  
`operator&`, 1

## P

pas de paràmetres, 3  
per referència constant, 3  
per valor, 3  
per valor constant, 3  
punter, 1  
punter genèric, 2  
punter local, 13  
punter nul, 2

## R

referència, 2  
Regla dels tres grans, 14  
representació, 69

## S

Segmentation fault, 2  
string, 48  
find, 50  
length, 49  
`operator+`, 49  
`operator+=`, 49  
replace, 50  
substr, 50

## T

taules, 6

## V

variable  
variable local, 95

- variable privada de classe, 94

- variables globals, 93

- vector, 51

- `void*`, *Vegeu* punter genèric