C++ orientat a objectes

Josep Maria Ribó

setembre 1999

Índex

1 Les classes **5**

1.1 Concepte . . . 5

1.2 Els elements i l'accés . . . 5

1.2.1 Els membres . . . 6

1.2.2 La definició dels membres . . . 6

1.2.3 La creació d'instàncies . . . 7

1.2.4 L'accés als membres . . . 7

1.3 Els objectes de la casa . . . 9

1.4 Referències . . . 9

2 Els membres constructors **10**

2.1 Concepte . . . 10

2.2 Exemple 1 . . . 10

2.3 Sobrecàrrega de constructors . . . 11

2.4 Valors per defecte . . . 11

2.5 Els inicialitzadors . . . 12

2.6 Constructors i classes agregades . . . 12

2.7 Creació d'objectes constants . . . 14

2.8 Constructors per defecte . . . 14

2.9 Constructors copiadors . . . 14

2.9.1 Exemple 1 . . . 14

2.9.2 Exemple 2 . . . 15

2.9.3 Exemple 3 . . . 15

2.9.4 Exemple 4 . . . 16

2.9.5 Assignació vs. inicialització . . . 17

2.9.6 Constructors copiadors per defecte . . . 17

2.10 Construcció dinàmica d'objectes . . . 18

2.10.1 Formes de construcció d'objectes . . . 18

2.10.2 Reserva dinàmica de memòria: la instrucció `new` . . . 18

2.10.3 Un exemple . . . 20

2.11 Detalls addicionals 21
2.12 Referències 21

3 Els membres destructors **22**
3.1 Concepte 22
3.1.1 Les deixalles 22
3.1.2 L'estratègia de C++ 24
3.1.3 Els membres destructors 24
3.1.4 La instrucció `delete` 24
3.2 Exemple 26
3.3 Creació i destrucció d'objectes estàtics 27
3.4 Creació i destrucció dinàmiques d'objectes 28
3.5 Creació i destrucció dinàmica de vectors d'objectes 29

4 Les classes derivades **32**
4.1 Concepte 32
4.2 Herència 33
4.2.1 Exemple 1 33
4.2.2 Exemple 2 35
4.2.3 Exemple 3 36
4.3 Subtipus 39
4.4 Els constructors en les classes derivades 40
4.4.1 Exemple 40
4.4.2 Exemple 2 41
4.5 Els destructors en les classes derivades 41
4.5.1 Un exemple 41
4.6 Els tipus de membres 43
4.7 Els tipus d'herència 44

5 El polimorfisme **48**
5.1 Exemple 1 48
5.2 Les funcions virtuals 53
5.2.1 Exemple 1 53
5.2.2 Exemple 2 56
5.2.3 La crida a les funcions virtuals 57
5.3 Les classes abstractes 61
5.4 Constructors i destructors virtuals 63
5.4.1 Constructors virtuals 63
5.5 Destructors virtuals 63
5.6 Exemple: La pila d'objectes de la casa 64

5.6.1 Primera aproximació . . . 64
5.6.2 La clonació . . . 64
5.6.3 La destrucció . . . 68
5.6.4 La pila d'objectes de la casa . . . 70
5.6.5 El client . . . 72

6 Les classes genèriques. Els `template` 73

6.1 Concepte . . . 73
6.2 Exemple 1 . . . 73
6.3 Exemple 2 . . . 75
6.4 Exemple 3 . . . 76
6.5 Exemple 4 . . . 77

7 Les funcions `friend` 79

8 La sobrecàrrega de funcions i operadors 82

8.1 Concepte . . . 82
8.2 Sobrecàrrega de funcions . . . 83
8.3 Sobrecàrrega d'operadors . . . 83
8.3.1 Sobrecàrrega d'operadors amb membres de la pròpia classe . . . 83
8.4 Sobrecàrrega d'operadors amb funcions *friend* . . . 84
8.5 La sobrecàrrega de l'operador << . . . 86
8.6 Detalls addicionals . . . 87

9 E/S en C++ 88

9.1 Introducció a l'E/S en C++ . . . 88
9.1.1 Fluxos . . . 88
9.1.2 Operacions bàsiques sobre fluxos . . . 89
9.1.3 Jerarquia de classes . . . 92
9.2 Fluxos d'entrada (classe *istream*) . . . 93
9.3 Fluxos de sortida (classe `ostream`) . . . 95
9.4 Formats . . . 95
9.5 Estats . . . 96
9.6 Treball amb fitxers: Les classes *ifstream, ofstream* i *fstream* . . . 97
9.6.1 L'operació *open* . . . 98
9.6.2 Exemples de treball amb fitxers . . . 99
9.6.3 Fitxers binaris . . . 103
9.7 La classe *FitxerSequencial* . . . 104

A Referències. Pas de paràmetres per referència 107

A.1 Què són les referències? . . . 107

A.2 Les referències constants 108
A.3 El pas de paràmetres per referència 108
A.3.1 Motivació 108
A.3.2 Com es fa el pas de paràmetres per referència en C++? 111
A.3.3 Les referències constants i el pas de paràmetres 111
A.3.4 Quan s'ha de fer un pas de paràmetres per referència en C++? 112

B Compilació separada **113**
B.1 Fitxers de capcelera, fitxers d'implementació i relacions *usa* 113
B.1.1 Fitxers de capcelera i d'implementació 113
B.1.2 Relacions *usa* 117
B.1.3 Compilació condicional 118
B.2 Procés de compilació 119
B.3 Automatització del procés de compilació 120

Capítol 1

Les classes

1.1 Concepte

Una classe és una agrupació d'objectes que comparteixen la mateixa estructura i comportament.

La definició de la classe s'encarrega precisament de descriure aquella estructura i comportament.

Anomenarem *membre* d'una classe a qualsevol element que formi part de la definició d'aquella classe, ja sigui una operació de la classe o bé un atribut de la mateixa. Les operacions d'una classe descriuen el seu comportament mentre que els atributs, la seva estrutura.

A cadascun dels objectes que comparteixen l'estructura i el comportament de la classe l'anomenarem *instància* d'aquella classe.

Les classes de C++, en general, consten d'una part privada que contindrà tots aquells elements (o *membres*) de la classe que vulguem ocultar als clients i una part pública amb tots els membres que la classe deixa veure a l'exterior.

En realitat, les classes en C++ no són més que un enriquiment de les *structs* de *C*, a les que incorporen la possibilitat de definir una part privada no visible des de l'exterior i també la possibilitat de definir funcions com a membres de la classe.

1.2 Els elements i l'accés

Considerem la següent classe que podríem utilitzar per a representar el concepte `data`:

```
class data {

 int dia;
 char mes[4];
 int any;

public:
 void fer_data(int d, char* m, int a){dia=d; strcpy(mes,m); any=a;}
 int quin_dia(){return dia;}
 char* quin_mes(){
        char* aux;
        aux=new char[4];
        strcpy(aux,mes);
```

```
        return aux;
 }
 int quin_any(){return any;}
};
```

La instrucció `aux=new char[4]` reserva un vector de 4 `char` de la memòria dinàmica i fa que l'apuntador `aux` referencï el primer. En *C* faríem això mateix de la següent manera: `aux=malloc(4*sizeof(char))`. La instrucció `new` es descriu amb tot detall a la secció 2.10.2.

1.2.1 Els membres

A la definició de la classe podem observar les dues parts a les què fèiem referència:

- La primera part, que defineix els membres (atributs) `dia, mes` i `any`, és la part *privada* de la classe i ningú des de fora de la mateixa no hi pot accedir [1]. Per defecte, tots els membres d'una classe són privats.
- La segona part, que defineix les operacions que es podran aplicar sobre la classe, és la part *pública*. Aquesta part es caracteritza perquè és visible des de qualsevol funció externa a la classe. Per aquest motiu es diu que els mètodes `fer_data, quin_dia, quin_mes i quin_any` són la interfície de la classe, o sigui, la finestra mitjançant la qual la classe es comunica amb l'exterior.

Referències:

A la secció 4.6 hi ha més detalls sobre la visibilitat dels membres de la classe i a 7 es presenten les funcions *friend*. A 2.10.2 es descriu la instrucció `new`.

1.2.2 La definició dels membres

Tal com s'estableix a 1.1, podem distingir dos tipus de membres en una classe: els *atributs* i les *operacions*.

Atributs: Constitueixen l'estructura de la classe i es defineixen a la part pública o privada de la forma usual.(Ex. `int dia;`)

Operacions: Descriuen el comportament d'aquella classe i es declaren sempre a la part pública o privada de la classe en la forma usual (Ex. `int quin_dia();`).

El seu codi es defineix:

- Immediatament després de la declaració:
  `int quin_dia(){return dia;}`
  Optarem per aquesta solució si el codi és molt senzill o s'ha de cridar molt sovint
- Fora de la definició de la classe:

```
class data {

 int dia;
 char mes[4];
 int any;

public:
 void fer_data(int, char*, int);
```

[1] Hem de fer aquí l'excepció de les anomenades funcions *friend* que es descriuen a la secció 7.

```
 int quin_dia();
 int quin_mes();
 int quin_any();
};

void data::fer_data(int d, char* m, int a)
{
  dia=d;
  strcpy(mes,m);
  any=a;
}

int data::quin_dia()
{
  return dia;
}

(...)
```

1.2.3 La creació d'instàncies

Un cop definida la classe *data*, podem considerar instàncies d'aquesta classe. Això ens porta a la definició d'objectes.

Un objecte de la classe `data` es defineix:

`data d1;`

En el moment de la definició de l'objecte es reserva un espai a la memòria per a encabir-lo. És en aquest moment que es crea l'objecte. Un objecte comença a existir a partir del moment en què se li assigna un espai en memòria diferent al dels altres objectes. És en aquest moment que es dota l'objecte d'una *identitat*.

D'igual manera podríem definir objectes referents a la classe *data* (i.e. apuntadors a objectes de la classe *data*).

`data* pd1;`

Aquesta definició **no** crea un objecte de tipus `data` sinó, més aviat, un *objecte referent* a *data*. Dit d'una altra manera, aquesta instrucció no provoca la reserva d'un espai físic per a un objecte de tipus *data*.

Per a crear un objecte de tipus *data* referenciat per `pd1`, fem el següent:

`pd1= new data;`

A 2.10.2 s'explora amb més detall aquesta manera de crear objectes dinàmicament.

Finalment, en l'àmbit de la creació d'objectes cal fer especial esment a la possibilitat d'inicialitzar objectes en el moment de la seva creació. D'això se n'encarreguen els membres constructors que seran estudiats en detall a la secció 2

1.2.4 L'accés als membres

Un dels elements fonamentals de la programació orientada a objectes (POO) és el de considerar un objecte x de classe A com a propietari d'una sèrie d'atributs i operacions (definides a la interfície de A). Alguns d'aquests atributs i operacions estaran definits a la part pública de la classe i constituiran la seva interfície. La resta d'objectes poden *modificar o consultar* l'estat de x, aplicant sobre ell les operacions

i atributs de la interfície de A.

Seguint aquesta idea, en C++, una funció qualsevol pot utilitzar els membres que defineix la classe `data` aplicats a l'objecte `d1` d'aquesta classe, de la següent manera [2]:

```
d1.fer_data(10,"nov",90);
```

Aquesta és una crida a l'operació membre *fer_data* aplicada sobre l'objecte `d1`. Aquesta operació provocarà la inicialització de l'objecte sobre el qual s'aplica (i.e. `d1`) al valor 10-nov-90.

Pel que fa a l'objecte referent `pd1`, podem treballar amb ell com segueix:

```
pd1=&d1;
(*pd1).fer_data(10,"nov",91);
```

Aquesta operació provoca una nova inicialització del mateix objecte `d1` (ara amb la data 10-nov-1991) ja que `*pd1` és un sinònim de `d1`.

La darrera crida admet la simplificació següent:

```
pd1->fer_data(10,"nov",91);
```

De la mateixa manera podríem cridar altres membres de la classe *data* declarats a la seva part pública:

`d1.quin_any()` retornaria 90.

`cout << d1.quin_dia();` escriuria a la sortida estàndar el valor 10.

En canvi, la crida a membres de la classe *data* definits a la seva part privada només és vàlida dins d'una operació de la classe:

`temp=d1.dia;` (només vàlid des de dins d'una operació de la classe `data` perquè `dia` és un membre *privat*).

A voltes no cal prefixar la crida a un membre d'una classe per l'objecte sobre el qual es fa aquella crida. Recordem, per exemple, el codi de l'operació `quin_dia()` de la classe *data*. La crida `d1.quin_dia();` executarà el codi `{return dia;}` (on el membre `dia` no va prefixat per res). En aquest cas s'entén que la funció `quin_dia` ha de retornar el valor de l'atribut *dia* de l'objecte $d1$ (que és aquell sobre el qual s'ha cridat la funció `quin_dia()`). Com a conseqüència, la crida `d1.quin_dia();` retornarà el valor $d1.dia$ (i.e. 10).

Com a resum, podem establir el següent:

> Un objecte z d'una classe B pot canviar o consultar l'estat d'un altre objecte x d'una classe A aplicant sobre x un atribut o operació definits públics a la classe A (interfície de A).
>
> Per aplicar un membre públic f (atribut o operació) d'una classe A sobre un objecte x d'aquella classe A, cal prefixar el nom del membre amb el de l'objecte sobre el qual s'aplica:
>
> `x.f(...);`
>
> Si la crida anterior es fa dins d'una operació g de la classe A i l'objecte sobre el qual es vol aplicar f coincideix amb l'objecte sobre el qual s'ha aplicat g, es pot fer la crida a f sense prefix.

```
class A{

public:

f(...);
g(...){... f(...);} //En la crida a f s'enten que l'objecte sobre el qual
    //es vol aplicar f es el mateix que l'objecte sobre el
```

[2] A la figura 2.1 es presenta gràficament les diferències entre la creació de `d1` i `pd1`

```
        //qual s'ha aplicat g.

    };
```

Finalment, una forma de referir-se dins el codi d'una operació de la classe a l'objecte sobre el qual s'ha fet la crida a aquella operació és mitjançant el referent `this`. Així,

`return dia;` és equivalent a `return this->dia;`

1.3 Els objectes de la casa

Tot seguit presentem un altre exemple de creació d'una classe en C++. Aquest exemple serà especialment rellevant perquè hi farem referència durant tot el text.

```
class objecte_casa {

  char lloc[MAXC];
  char marca[MAXC];
  char nom[MAXC];
  int preu;
public:
  void crear_objecte_casa(char* plloc, char* pmarca,
                          char* pnom, int ppreu);
  void llista_caract();
  };
```

Aquest exemple presenta la creació de la classe *objecte_casa* amb quatre atributs:el lloc de la casa on es guarda l'objecte, la marca que el comercialitza, el seu nom i el preu.

A la part pública proveïm, com a interfície de la classe, una operació creadora d'un objecte de la casa i una altra que permet la visualització dels seus atributs.

1.4 Referències

- A 7 es presenten les funcions *friend.*
- A 2 s'estudien els membres constructors.

Capítol 2

Els membres constructors

2.1 Concepte

Algunes de les funcions membres d'una determinada classe tenen com a objectiu inicialitzar objectes d'aquella classe (és el cas de les accions `crear_data` i `crear_objecte_casa`). Aquestes funcions les cridem immediatament després d'haver creat l'objecte. Això pot donar lloc a errors de no inicialització d'objectes, en el cas que ens oblidem de cridar-les. Una idea que podria resoldre aquest problema i dotar el programa d'una major elegància fóra la d'inicialitzar automàticament els objectes en el moment de la seva creació. Per fer això utilitzarem unes funcions membres especials que anomenarem *constructors.*

Els *constructors* són funcions membres d'una classe tals que:

- Tenen el nom de la classe, no tenen tipus de retorn i, opcionalment, prenen alguns paràmetres que serviran per a inicialitzar l'objecte adequadament.
- Es criden implícitament i automàtica en el moment de la creació de l'objecte.
- La seva missió és la de completar la creació de l'objecte i inicialitzar-lo convenientment.

2.2 Exemple 1

Prenem l'exemple de la secció 1.2 i sutituïm la funció membre `fer_data` per una constructora que tingui els mateixos efectes. La definició de la classe *data* ens queda de la següent manera:

```
class data {

 int dia;
 char[4] mes;
 int any;

public:
 data(int d, char* m, int a){dia=d; strcpy(mes,m); any=a;}
 //Resta de les operacions de la classe data.
};
```

I la definició d'un objecte de tipus data és ara de la forma:

```
data d1(10,"nov",90);
```

En el moment de la definició de l'objecte (execució de la instrucció anterior) es cridarà automàticament la funció membre constructora `data` amb els paràmetres actuals pertinents (en aquest cas, 10, "nov" i 90).

2.3 Sobrecàrrega de constructors

Podríem tenir més d'un constructor per la mateixa classe (sobrecarregat) amb paràmetres diferents:

```
class data {

 int dia;
 char[4] mes;
 int any;

public:
 data(int d, char* m, int a){dia=d; strcpy(mes,m); any=a;}
 data(){dia=1; strcpy(mes,"gen"); any=0;}

 //Resta d'operacions de data
};

data d1(10,"nov",90), d2;
```

`d1` s'inicialitzarà a la data 10-"nov"-90 mentre que d2 s'inicialitzarà amb la data 1-"gen"-00.

Tota definició d'un objecte implica la crida implícita a algun dels constructors de la classe a la que pertany aquell objecte. Per tant,

- A la definició d'un objecte s'han d'incloure els paràmetres que necessita *algun* dels constructors de la classe de l'objecte.

  Exemple: `data d1(10,"nov",90);`

- Si es vol tenir la llibertat de definir un objecte sense donar-li cap tipus d'inicialització (i això té sentit en el context de la classe en la qual es defineix aquell objecte) es pot sobrecarregar un constructor sense paràmetres i sense codi:

  `data(){}`

- Si una classe no té cap constructor definit, se'n crea un d'implícit que és precisament el constructor *buit* (sense paràmetres i sense codi).

  `data(){}`.

2.4 Valors per defecte

Una altra possibilitat de tractar amb constructors és incorporar inicialitzacions per defecte als paràmetres de la pròpia definició del constructor:

```
data(int d=1, char* m="gen", int a=0) {dia=d; strcpy(mes,m); any=a;}
```

Aquest constructor es podria activar llavors de diferents maneres:

`data avui(12,"oct",96);`

`avui` s'inicialitza amb la data 12-"oct"-96

`data fa_temps;`

`fa_temps` s'inicialitza amb la data 1-"gen"-00

2.5 Els inicialitzadors

Finalment, existeix una construcció de C++ que s'utilitza força sovint amb els constructors:

```
data(int d, char* m, int a)
 :dia(d), any(a)
 {strcpy(mes,m);}
```

D'aquesta forma inicialitzem directament els membres `dia` i `any` als valors `d` i `a` respectivament[1]. A les expressions `dia(d)` i `any(a)` les anomenem *inicialitzadors*.

Si utilitzem inicialitzadors, pot donar-se el cas que el codi de l'operació constructora esdevingui buit. Un exemple ben senzill d'això l'obtenim canviant la definició de l'atribut `mes`:

```
class data2{

 int dia;
 int mes;
 int any;

public:

 data2(int d, int m, int a) :dia(d), mes(m), any(a) {}
 //Resta d'operacions
};
```

El codi de la constructora esdevé buit ja que l'única feina de la funció és la d'inicialitzar aquests membres.

La necessitat d'introduir aquesta nova sintaxi es fa palesa quan utilitzem classes agregades, és a dir, classes que tenen altres classes com atributs. D'aquesta possibilitat ens n'encarreguem a l'apartat següent.

2.6 Constructors i classes agregades

Considerem la classe persona definida de la manera següent:

```
class persona{
```

[1]En realitat, la sintaxi `dia (d)` correspon a la crida al constructor copiador de la classe de l'atribut `dia` segons s'explica a l'apartat 2.9.

```
  char nom[30];
  data data_naixem;
  char nif[9];

public:

 persona(char* pnom, char* pnif, int pdia, char* pmes, int pany)
    :data_naixem(pdia,pmes,pany)
 {
  strcpy(nom, pnom);
  strcpy(nif,pnif);
 }

 //altres operacions

};
```

Com la classe *persona* conté un membre d'una altra classe (*data*), és evident que quan creem una instància de la classe *persona* estarem creant, en particular, una instància de la classe *data* i, en conseqüència, dins del constructor de la classe *persona* haurem de cridar el constructor de la classe *data*. `:data_naixem(pdia,pmes,pany)` s'encarrega precisament de fer això.

`:data_naixem(pdia,pmes,pany)`, doncs, no és altra cosa que una crida explícita al constructor de la classe *data* per tal d'inicialitzar el membre *data_naixem*.

En primera instància pot semblar que aquesta construcció es pot sustituir per una simple assignació dins el codi del constructor de *persona*.

Exemple:

```
class persona{

char nom[30];
data data_naixem;
char nif[9];

public:

persona(char* pnom, char* pnif, data pdata)
{
  strcpy(nom, pnom);
  strcpy(nif,pnif);
  data_naixem=pdata;
}

//altres operacions

};
```

Per a que això funcioni són necessàries tres condicions:

- Ha d'existir un constructor per a la classe *data* sense paràmetres:
  `data()`
- Ha d'existir a la classe *data* un *constructor copiador* (veure 2.9) correctament definit (encarregat de dur a terme el pas del paràmetre `pdata` del constructor de *persona*).
  `data(data& d)...`

- Ha d'existir una definició correcta de l'operació d'assignació entre objectes de la classe *data* (encara que no sigui el cas de la classe *data*, convé recordar que l'assignació per defecte de C++ no funciona per aquelles classes tals que la seva representació incorpora memòria dinàmica perquè els objectes que s'assignen passen a compartir identitat).

Per acabar ens cal indicar que els inicialitzadors es poden col.locar únicament immediatament abans del codi d'un constructor i que un inicialitzador és en realitat *una crida explícita a un constructor de la classe de l'atribut que es vol inicialitzar* (i.e. `:data_naixem(pdia,pmes,pany)` és una crida a un constructor de la classe `data` i `dia(d)` és una crida a un constructor de la classe `int`).

2.7 Creació d'objectes constants

Els constructors d'una classe ens permeten crear objectes instàncies d'aquella classe *constants* (això és, que no poden ser modificats). La sintaxi que utilitzarem és la següent:

```
data const dc(10,"feb",90);
```

2.8 Constructors per defecte

Si una classe no té cap constructor definit, el compilador en defineix un per defecte sense paràmetres i amb codi buit.

2.9 Constructors copiadors

> Un *constructor copiador* per a una classe C és un constructor que té com a paràmetre una referència a un objecte de la classe C i, en conseqüència, es pot utilitzar per a inicialitzar l'objecte que s'està creant directament amb un valor de la classe C [2].

2.9.1 Exemple 1

```
class data {

 int dia;
 char mes[4];
 int any;

public:
 data(int d, char* m, int a) :dia(d),any(a){strcpy(mes,m);}
 data(){dia=1; strcpy(mes,m); any=0;}
 data(const data& pdata)
   :dia(pdata.dia),any(pdata.any)
   {strcpy(mes,pdata.mes);}

 //Resta d'operacions de data

};
```

[2] A l'apèndix A es presenta el pas de paràmetres en $C++$ i, en particular, el pas de paràmetres per referència.

El tercer constructor és un constructor copiador.

Amb la classe presentada es podrien definir objectes de dues maneres noves:

```
data d1(10,"des",90);

data d2=d1;  //activacio del constructor copiador.
     //creacio de l'objecte d2 amb el valor de d1.

data d3(d1); //activacio del constructor copiador.
     //creacio de l'objecte d3 amb el valor de d1.
```

Fem algunes anotacions respecte els constructors copiadors:

- No admeten com a paràmetre objectes de la classe del constructor sinó referències a objectes d'aquesta classe.
- La característica `const` associada al paràmetre del constructor copiador de l'exemple anterior no és imprescindible però resulta útil per dos motius:
  - Impideix que, per error, modifiquem el valor de l'objecte utilitzat per fer la inicialitació (un paràmetre amb la característica `const` no pot ser modificat dins de l'acció.
  - Permet inicialitzar un objecte d'una classe amb un objecte *constant* d'aquella classe (veure 2.7).

2.9.2 Exemple 2

Els constructors copiadors són els que s'encarreguen de la inicialització dels paràmetres de les funcions passats per valor.

```
void f(data d){...}

//..

main()
{
  data avui(10,2,90);

  //...

  f(avui);
  //...

}
```

El constructor copiador de *data* és l'encarregat de casar el paràmetre actual `avui` amb el formal `d`.

2.9.3 Exemple 3

Els *inicialitzadors* poden servir com un altre exemple d'activació dels constructors copiadors:

```
class persona{

 char nom[30];
 data data_naixem;
 char nif[9];

public:

 persona(char* pnom, char* pnif, data pdata)
   :data_naixem(pdata)
 {
  strcpy(nom, pnom);
  strcpy(nif,pnif);
 }

 //altres operacions

};
```

L'inicialitzador `:data_naixem(pdata)` és una manera explícita d'activar el constructor copiador que a l'exemple 1 hem definit per la classe *data*.

2.9.4 Exemple 4

Finalment les instruccions `return` també utilitzen constructors copiadors ja que la semàntica associada a la instrucció `return` indica que aquesta instrucció *inicialitza* una variable amb el valor que retorna la funció.

```
data dema(data& d)
{

  data aux(d.quin_dia()+1, d.quin_mes(), d.quin_any());

  return aux;

}

main()
{
  data avui(12,"abr",98);
  data dem;

  dem=dema(avui);   //dem s'inicialitza amb el valor que retorna
                    //la funcio dema. Aquesta inicalitzacio es fa
                    //mitjancant la crida a un constructor copiador.

}
```

Als exemples precedents hem vist quatre usos dels constructors copiadors:

1. Inicialització explícita d'objectes (exemple 1).
2. Pas de paràmetres per valor (exemple 2).
3. Retorn del valor d'una funció (exemple 4).
4. Inicialitzadors dins d'altres constructors (exemple 3).

2.9.5 Assignació vs. inicialització

C++ distingeix els dos conceptes.

L'assignació la duu a terme mitjançant la sobrecàrrega de l'operador d'assignació (=) tal com s'explica a 8 i s'utilitza per les assignacions habituals entre variables dels programes:

```
data d1(...), d2(...);

//...

d1=d2;
```

La inicialització s'aconsegueix amb l'ajut dels constructors copiadors i s'utilitza en:

- Assignació de valor inicial als objectes (en el moment de la seva creació):

  ```
  data d1(10,"abr",89);
  data d2=d1;           //inicialitzacio d2
  ```

  O bé

  ```
  persona (char* pnom, char* pnif, data& pdata)
  :data_naixem(pdata)
  {
  //...
  }
  ```

- Assignació de valor als paràmetres formals en un pas de paràmetres per valor.
- Assignació del valor de retorn d'una funció a una variable.

2.9.6 Constructors copiadors per defecte

Si no es proveeix cap constructor copiador, C++ en genera un per defecte, amb la característica que copia només els valors dels atributs de la classe a la qual pertany. Aquest fet planteja problemes de compartició d'identitat en el cas que la classe presenti part de la seva representació en memòria dinàmica accessible a través d'apuntadors.

2.10 Construcció dinàmica d'objectes

2.10.1 Formes de construcció d'objectes

Hi ha dues formes de construir objectes:

- Construcció estàtica.
- Construcció dinàmica.

La primera forma és la que hem utilitzat a tots els exemples presentats. Consisteix en associar una classe a un nom de variable. El compilador reserva en memòria l'espai necessari per a crear un objecte d'aquella classe amb el nom donat. En temps d'execució es crida el constructor de la classe per a inicialitzar l'objecte.

Exemple: `data avui(10,"nov",90);`

La visibilitat i el període de vida d'un objecte creat d'aquesta manera coincideixen amb l'àmbit de l'acció o funció allà on es defineix. Quan s'acaba l'execució d'aquella acció, l'objecte deixa d'existir.

Si l'objecte es crea fora de l'àmbit d'una acció o funció, aleshores és visible des de qualsevol lloc del fitxer allà on es defineix a partir del punt de la seva definició. El seu període de vida abarca des del moment de la seva creació fins a la fi de l'execució del programa.

En la segona forma, tot el procés de creació de l'objecte (tant la reserva d'espai en memòria com la crida al constructor) es fa en temps d'execució i utilitzant la memòria dinàmica. Per això cal disposar d'una manera explícita de reservar memòria (la instrucció `new` s'encarega d'això).

Exemple:

```
data* pd1;

pd1=new data(10,"nov",90);
```

El període de vida dels objectes creats d'aquesta forma ultrapassa l'àmbit de l'acció allà on s'han definit. Cal destruir-los explícitament amb la instrucció `delete` (veure més detalls a 3).

En aquest apartat ens ocuparem d'aquesta segona forma de creació d'objectes: *la creació dinàmica d'objectes* .

2.10.2 Reserva dinàmica de memòria: la instrucció `new`

La instrucció `new` permet construir un objecte d'una classe T dinàmicament reservant la porció de memòria necessària per a encabir aquell objecte. `new` retorna un apuntador a aquella porció de memòria. En la seva forma més simple s'utilitza de la següent manera:

```
T* pt;

pt=new T;
```

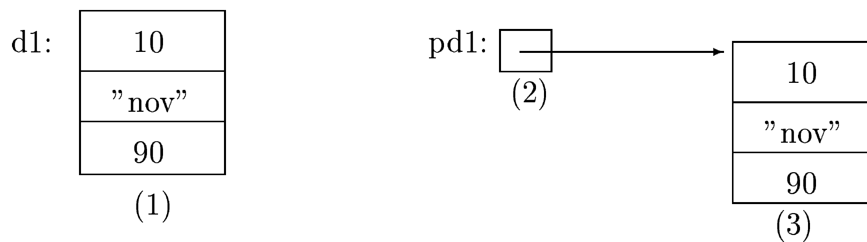
On T és el nom d'una classe o d'un tipus predefinit.

pt és un referent a l'objecte de tipus T creat dinàmicament per `new`.

data d1(10,"nov",90);

data* pd1;

pd1=new data(10,"nov",90);



(1): Objecte instància de la classe *data*.
(2): Objecte referent a la classe *data*.
(3): Objecte anònim de la classe *data*.

Figura 2.1: Formes de construcció d'un objecte.

Tot seguit presentem alguns detalls addicionals respecte `new`:

new **crida un constructor de la classe**

Si l'objectiu de `new` és el de crear un nou objecte d'una determinada classe T, necessàriament haurà de cridar implícitament algun dels constructors d'aquella classe.

new **pot prendre paràmetres**

Si `new` ha de cridar algun constructor d'una classe, sembla clar que haurà de poder prendre els paràmetres que aquell constructor necessiti. Així doncs, `pd1=new data(10,"nov",90);` crea un nou objecte de classe *data*, referenciat per `pd1` i l'inicialitza a la data 10-"nov-90 cridant el constructor corresponent d'aquesta classe.

Construcció de vectors dinàmics

La construcció `new` permet també definir vectors en temps d'execució. Aquests vectors no tindran la restricció de la mida màxima fixada en temps de compilació. El següent exemple defineix un vector de `n` enters:

```
int* pvint;
int n;

//inicialitzacio de n...

pvint=new int[n];
```

No es pot combinar en una sola instrucció `new` la creació dinàmica d'un vector d'elements de classe T i la inicialitació d'aquells elements mitjançant la crida a un constructor de T amb paràmetres.

2.10.3 Un exemple

A voltes, en la creació d'un objecte d'una determinada classe, combinem les dues formes de construcció (estàtica i dinàmica) que hem vist. En aquest apartat en descrivim un exemple.

Suposem que introduim la classe *medicament* com una classe semblant a *objecte_casa*[3]. Els atributs rellevants de la classe *medicament* seran el *nom, el lloc, la marca, el preu, el metge que l'ha receptada i quina malaltia cura.* Una forma adient de veure aquests atributs és com a cadenes de caràcters, però suposem que aquest cop volem representar aquestes cadenes de caràcters en memòria dinàmica. Aquest propòsit ens portarà a una definició de la classe *medicament* com la següent:

```
class medicament{

  char* nom;              //en lloc de char nom[MAX];
  char* lloc;             //en lloc de char lloc[MAX];
  char* marca;            //en lloc de char marca[MAX];
  int preu;
  char* metge;            //en lloc de char metge[MAX];
  char* malaltia;         //en lloc de char malaltia[MAX];

public:
  medicament(char* pnom, char* plloc, char* pmarca, int ppreu,
     char* pmetge, char* pmalaltia)
  {
     nom=new char[strlen(pnom)+1];
     lloc=new char[strlen(plloc)+1];
     marca=new char[strlen(pmarca)+1];
     metge=new char[strlen(pmetge)+1];
     malaltia=new char[strlen(pmalaltia)+1];

     strcpy(nom,pnom);
     strcpy(lloc,plloc);
     strcpy(marca,pmarca);
     preu=ppreu;
     strcpy(metge,pmetge);
     strcpy(malaltia,pmalaltia);
  }

//Altres operacions

};
```

La creació d'un objecte de classe *medicament*, la fem de la forma usual:

```
  medicament x1("contradol", "farmaciola", "vayer", 500,
"Dr. Blanch", "mal de cap");
```

El funcionament d'aquesta instrucció és el següent: en temps de compilació es reservarà espai en memòria suficient per a emmagatzemar un enter i 5 apuntadors a caràcter, mentre que en temps d'execució, la

[3]En realitat, a l'apartat 4 veurem que podem considerar la classe *medicament* com una *classe derivada* d'*objecte de la casa.*

crida al constructor definit, acabarà de reservar (dinàmicament) la resta de l'espai que ocuparà finalment `x1`. A la figura 3.2 es presenta l'objecte `x1` un cop construït.

2.11 Detalls addicionals

Un constructor no pot ser:

- `const`
- `volatile`
- `virtual`
- `static`

Un constructor es pot utilitzar explícitament:

```
main()
{
data d;    //suposem existencia cosntructor sense parametres.

//...

data(1,"nov",98);    //creacio objecte anonim

d=data(1,"nov",98);

//...


}
```

2.12 Referències

Aquest apartat ha estat una introducció als constructors. En els propers apartats completarem la presentació dels mateixos.

- La relació entre els constructos, els destructors i la memòria dinàmica es presenta a 3.
- La utilització dels constructors en la construcció de classes derivades i la seva relació amb l'herència queda palesa a 4.4.
- Les particularitats dels constructors relacionades amb el polimorfisme es presenten a 5.4.1.

Capítol 3

Els membres destructors

3.1 Concepte

De la mateixa manera que les classes defineixen membres constructors d'objectes, sovint necessiten també definir membres destructors. Per a justificar-ne la necessitat, hem de presentar en primer lloc què són les deixalles.

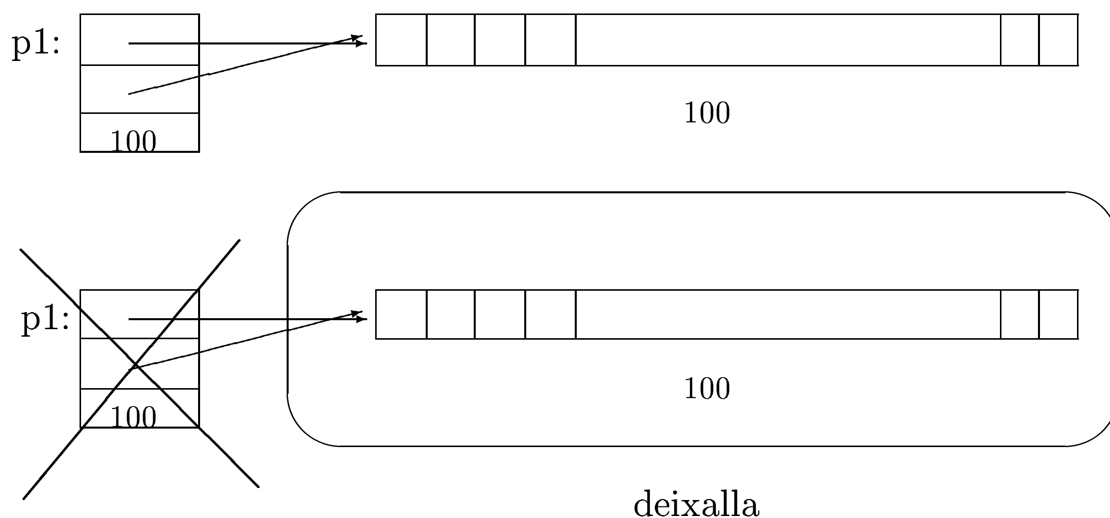
3.1.1 Les deixalles

Sovint, en la construcció d'un objecte, s'utilitza memòria reservada dinàmicament. Posteriorment, quan acaba l'àmbit de definició d'aquell objecte, si no es fa res per a impedir-ho, aquell espai de memòria dinàmica utilitzat per a emmagatzemar una part del objecte es pot convertir en una *deixalla*.

> Les deixalles són fragments de memòria dinàmica que són inaccessibles i, alhora, no reutilitzables.

Un exemple de creació de deixalles podria ser el següent:

```
class pila_car{

  char* pila;
  char* cim;
  int tamany;

public:

  pila_car(int tam){tamany=tam; pila=new char[tam]; cim=pila;}
  //ALTRES OPERACIONS


};

void f()
{
  pila_car p1(100);
```



(1): Objecte $p1$ immediatament després de ser creat.
(2): Al final de l'acció f, la part estàtica de $p1$ es destrueix però no la seva part dinàmica, la qual es converteix en deixalla.

Figura 3.1: Generació d'una deixalla

```
  //codi de f
}

main()
{
  f();
 //Mes coses
}
```

L'acció f s'encarrega de reservar un espai de memòria dinàmica que esdevindrà inaccessible quan acabi l'execució de l'acció f (moment en què $p1$ deixarà d'existir) tal com es mostra a la figura 3.1.

Els ecologistes (enginyers de sistemes) ens adverteixen de les greus conseqüències que la progressiva degradació del medi (la memòria) tindrà en les nostres vides (els nostres programes) i, en conseqüència, ens demanen una sensibilització davant del problema i un compromís que porti a una política de reciclatge real i de posterior reutilització de les deixalles generades. Hi ha dues maneres de dur a terme aquesta política:

1. Periòdicament un procés anomenat *recollidor de deixalles* (*garbage collector*) s'encarrega de detectar i alliberar totes les deixalles que hi ha a la memòria. Aquesta és l'aproximació que prenen alguns entorns de programació orientats a objectes com ara el *smalltalk*. El problema d'aquesta aproximació és que el rendiment del sistema baixa com a conseqüència de les execucions periòdiques del programa recollidor de deixalles.

2. El propi programador allibera l'espai de memòria dinàmica que no necessitarà més abans no es converteixi en una deixalla. Aquesta és la política que pren el llenguatge de programació C. Aquesta aproximació resulta molt més eficient, alhora que de més baix nivell ja que el programador no pot fer abstracció de la gestió de la memòria.

3.1.2 L'estratègia de C++

C++ adopta una estratègia intermitja: és el programador qui allibera l'espai inútil associat a un objecte però ho fa en forma d'operació destructora d'aquell objecte. Aquesta operació es crida de forma automàtica, en el moment en què acaba la vida del mateix. A l'exemple anterior, en acabar l'execució de l'acció f (moment en què acaba l'àmbit de definició de $p1$), s'hauria cridat a una operació destructora de l'objecte $p1$ que hauria evitat la generació de la deixalla.

3.1.3 Els membres destructors

Els membres destructors són operacions d'una classe que:

- Tenen el mateix nom de la classe precedit per " $\sim$ ", no prenen paràmetres ni retornen valors.
- Es criden implícitament (automàticament) quan cal destruir un objecte que altrament es convertiria en una deixalla. El seu codi explica què cal fer per a procedir a la liquidació d'aquell objecte. Habitualment al seu codi hi ha alguna instrucció d'alliberament de memòria. La instrucció que permet alliberar explícitament memòria dinàmica en C++ és `delete`.

L'exemple anterior quedaria de la següent manera afegint-hi ara una operació destructora:

```
class pila_car{

  char* pila;
  char* cim;
  int tamany;

public:

  pila_car(int tam){tamany=tam; pila=new char[tam]; cim=pila;}
  ~pila_car(){delete [] pila;}
  //ALTRES OPERACIONS


};
```

En fer `pila=new char[tam];` s'anota el tamany reservat per a l'apuntador *pila*. En fer `delete [] pila;`, el tamany anotat per aquest apuntador serà precisament el que s'alliberarà.

3.1.4 La instrucció `delete`

A l'exemple de l'apartat anterior hem utilitzat la instrucció `delete`. Però quin és exactament el seu sentit?

La instrucció `delete` serveix per a alliberar una porció de memòria que havia estat reservada amb la instrucció `new` i evitar, d'aquesta manera, que aquella porció de memòria esdevingui una deixalla. Com a conseqüència d'això, tindrà sentit d'utilitzar aquesta instrucció en dues situacions ben diferents:

- Quan acaba l'àmbit de definició d'un objecte definit estàticament.

```
class A{

//Representacio

public:

A(...);
~A();
//altres operacions

};

void f()
{

  A x1(...);

  //...

}
```

En aquest exemple, el final de l'acció f marcaria la fi de l'àmbit de definició de $x1$, objecte de la classe A.

Aquest és el cas que hem il.lustrat a l'exemple anterior. En ell convé que es generi una crida implícita al destructor de la classe A per tal d'evitar que, en acabar l'àmbit de $x1$ es doni lloc a una deixalla.

- Quan es desitja destruir explícitament un objecte creat dinàmicament (i alliberar la memòria que ocupa). Heus ací un exemple:

```
A* px1;

//...

px1=new A(...);

//...

delete px1;
```

La instrucció `delete` contempla dues formes diferents segons l'objecte que es vol destruir:

1. Es vol destruir un objecte *simple* de classe A (i.e. un objecte creat fent `px1=new A(...)`)

   En aquest cas n'hi haurà prou amb fer `delete px1`.

```
A* px1;
```

```
//...

px1=new A(...);

//...

delete px1;
```

L'expressió `delete px1;` allibera l'espai de memòria reservat per `px1=new A(...);` i crida al destructor de la classe *A* (si en té), ja que és possible que el constructor de la classe *A* que ha cridat (`px1=new A(...)`) hagi reservat alguna posició de memòria dinàmica que ara es convertiria en deixalla.

Per a veure-ho més clar, als següents apartats es desenvolupa un exemple.

2. Es vol destruir un *vector dinàmic d'objectes de classe A* (i.e. un vector creat fent `px1=new A[n]`).

```
A* px1;
int n;

//inicialitzacio de n i altres coses...

px1=new A[n];

//...

delete [] px1;
```

`delete [] px1;` destrueix cadascun dels elements del vector `px1` (que són de classe *A*). Si la classe *A* té destructor associat, aquest es crida per a cada element del vector que es destrueix.

Al següent apartat es presenta un exemple que esperem pugui resultar esclaridor.

3.2 Exemple

Recuperem la classe *medicament* introduïda a 2.10.3 i afegim-li ara un destructor:

```
class medicament{

  char* nom;
  char* lloc;
  char* marca;
  int preu;
  char* metge;
  char* malaltia;

public:
  medicament(char* pnom, char* plloc, char* pmarca, int ppreu,
```

```
     char* pmetge, char* pmalaltia)
  {
     nom=new char[strlen(pnom)+1];
     lloc=new char[strlen(plloc)+1];
     marca=new char[strlen(pmarca)+1];
     metge=new char[strlen(pmetge)+1];
     malaltia=new char[strlen(pmalaltia)+1];

     strcpy(nom,pnom);
     strcpy(lloc,plloc);
     strcpy(marca,pmarca);
     preu=ppreu;
     strcpy(metge,pmetge);
     strcpy(malaltia,pmalaltia);
  }

  ~medicament()
  {
    delete [] nom;
    delete [] marca;
    delete [] lloc;
    delete [] metge;
    delete [] malaltia;
  }

};
```

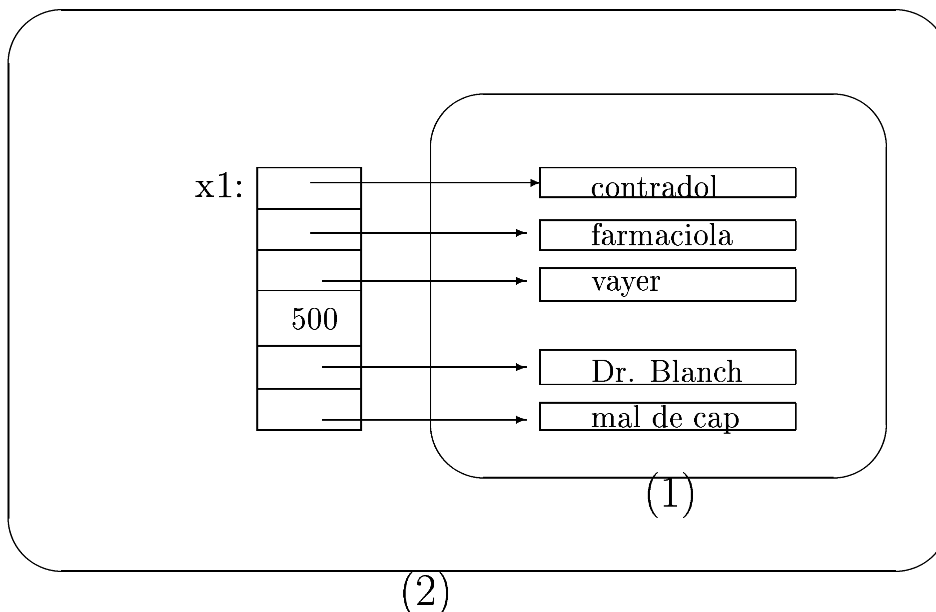
A partir d'aquesta classe podem il.lustrar tres formes d'actuació dels destructors:

3.3 Creació i destrucció d'objectes estàtics

La forma més usual de treball amb els constructors i destructors és la definició d'un objecte en un àmbit de declaració i la seva posterior destrucció a la fi d'aquell àmbit. Veiem-ho:

```
void f()
{

  medicament x1("contradol", "farmaciola", "vayer", 500,
"Dr. Blanch", "mal de cap");


  //fer coses

}
```

En la definició de $x1$ s'activa el constructor, mentre que, quan acaba el bloc on s'ha definit (l'acció f, en aquest cas) s'activa automàticament el destructor, el qual s'encarrega d'alliberar tota la memòria reservada dinàmicament per a poder encabir tots els camps de la classe. Aquest procés es presenta a la figura 3.2.



(1): Creat pel constructor de la classe medicament (mitjançant la instrucció `new`) i destruït pel destructor de la mateixa classe (mitjançant la instrucció `delete`). Sense l'acció del destructor aquesta part esdevindria una deixalla.
(2): Totalitat de l'objecte `x1`.

Figura 3.2: Creació i destrucció d'objectes

3.4 Creació i destrucció dinàmiques d'objectes

Una segona possibilitat de treball amb constructors i destructors és la creació dinàmica d'objectes (això és, en temps d'execució). Veiem-ne un exemple:

```
void f()
{
  medicament* px1;


  px1= new medicament ("contradol", "farmaciola", "vayer", 500,
       "Dr. Blanch", "mal de cap");


  //fer coses

  delete px1;
}
```

En aquesta acció *f*,

- creem, amb la instrucció `new`, un objecte *anònim* de la classe *medicament* en temps d'execució. Seguint la sintaxi de `new` discutida a 2.10.2, afegim a l'expressió els paràmetres que requereix el constructor de *medicament*.

- `new` crida dinàmicament el constructor de la classe *medicament* i reserva dinàmicament espai per a encabir els atributs.
- Fem que l'objecte referent $px1$ es refereixi a aquest objecte *medicament* que hem creat.
- A la fi de l'acció destruim explícitament, amb la instrucció `delete` l'objecte creat. En concret, la instrucció `delete` s'encarrega de:
  - Alliberar l'espai de memòria reservat per `new...` i
  - Cridar el destructor de la classe *medicament* que alliberarà l'espai de memòria reservat dinàmicament pel constructor d'aquella classe (i.e. els vectors dinàmics: `nom, marca, lloc, malaltia...`).

  Aquest procés s'esquematitza a la figura 3.3.

A la figura 3.4 es mostra com quedaria la memòria després de l'execució de `delete px1;` si *medicament* no tingués destructor definit o si els dissenyadors de C++ no haguessin imposat que l'operador `delete` cridés el destructor de la classe a la que pertany l'objecte que es destrueix.

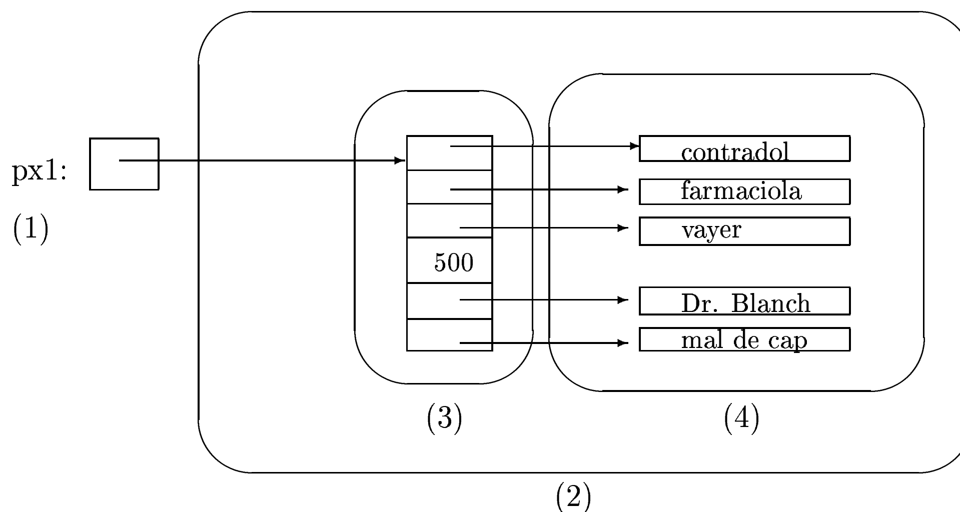
Els aspectes més importants que cal fixar del procés que hem descrit són els següents:

- `delete` no només allibera la porció de memòria que `new` ha assignat sinó que, a més a més, crida el destructor de la classe *medicament* per a que aquest alliberi, a la seva vegada, les porcions de memòria que va assignar dinàmicament el constructor de la classe (i.e. els vectors *dinàmics* `nom, marca, lloc, malaltia,....`).
- El període de vida dels objectes creats dinàmicament s'allarga fins a la seva destrucció explícita mitjançant l'expressió `delete` i no té, en conseqüència, res a veure amb el bloc (o àmbit) en el que s'han creat.
- La creació d'un objecte referent a *medicament* (*apuntador a medicament*) com ara `px1` no significa de cap manera la creació d'un objecte de classe *medicament*. En particular, la sentència `medicament* px1;` no involucra de cap manera una crida al constructor de la classe *medicament*.

3.5 Creació i destrucció dinàmica de vectors d'objectes

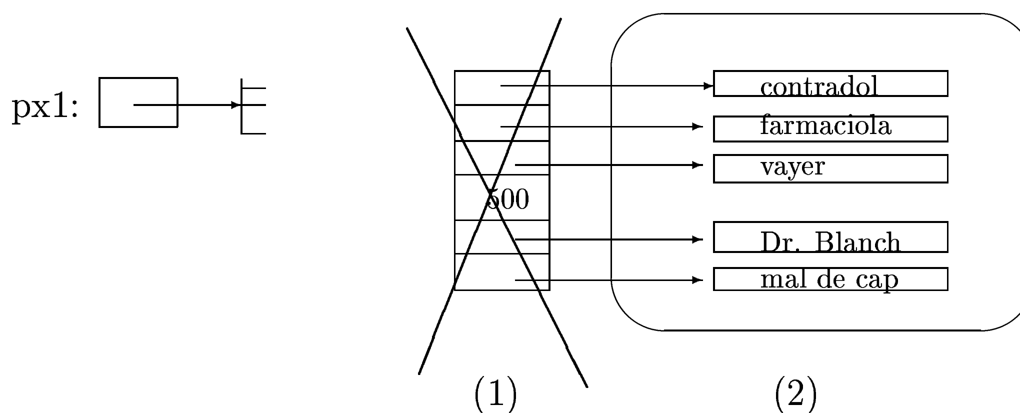
Suposem que volem crear dinàmicament un vector de *medicaments*. Per a poder fer aquesta operació utilitzant la instrucció `new` necessitem dotar la classe *medicament* d'un constructor sense paràmetres ja que, com hem vist a 2.10.2, no podem crear i inicialitzar al mateix temps un vector d'objectes amb l'expressió `new`. Fem-ho:

```
class medicament{

//Representacio

public:

medicament(){}     //Constructor sense parametres afegit

//Resta de constructors, destructor i operacions.

};
```



(1): Objecte referent a un objecte de la classe *medicament*
(2): Objecte anònim de la classe *medicament* referit per `px1`.
(3): Espai reservat per la instrucció `px1=new medicament;` i eliminat per la instrucció `delete px1;`.
(4): Espai reservat pel constructor de la classe *medicament* que és activat per la instrucció `px1=new medicament;` despres de reservar l'espai (3).
(4) És eliminat pel destructor de la classe `medicament` activat per `delete px1;`.

Figura 3.3: Creació i destrucció dinàmiques d'objectes.



(1): Espai eliminat per `delete px1;`
(2): Deixalles.

Figura 3.4: Actuació de la instrucció `delete px1;` si no hi hagués destructor definit per la classe *medicament*

```
void f()
{

  medicament* vmed;

  vmed=new medicament[10];


  vmed[0]= medicament ("contradol", "farmaciola", "vayer", 500,
       "Dr. Blanch", "mal de cap");
  vmed[1]= medicament (...);

  //...

   delete [] vmed;

}
```

Els aspectes més rellevants d'aquest exemple són els següents:

- L'expressió `new` crea dinàmicament un vector de 10 objectes de classe *medicament*. Per a crear-los crida al constructor sense paràmetres de la classe (el que hem afegit en aquest exemple).
- L'expressió `delete [] vmed;` s'encarrega de destruir tot el vector de medicaments i de cridar, per cada element del vector, al destructor de la classe *medicament* (que li permet alliberar els seus atributs *nom, marca, malaltia...* i evitar que es converteixin en deixalles).
- Les assignacions `vmed[i]= medicament(...)` són crides explícites al constructor de la classe *medicament*.

Capítol 4

Les classes derivades

4.1 Concepte

El fonament de les classes en un llengutge orientat a objectes és el de modelar els conceptes del problema que volem resoldre. Ara bé, en tot problema de complexitat mitjana, els conceptes no es presenten aïllats sinó que s'entrelliguen entre sí donant lloc sovint a *jerarquies de conceptes*. Suposem un problema en el que calgués gestionar els models de vehicles que fabrica una determinada multinacional del sector. Possiblement hauríem de tractar amb conceptes tals com *camions, furgonetes, turismes, esportius, utilitaris, familiars....* Tots aquests conceptes estan relacionats: els esportius, els utilitaris i els familiars són tots *turismes*, mentre que els camions, les furgonetes i els propis turismes els podríem englobar sota un terme més genèric com ara: *vehicles terrestres amb motor d'explosió* o, si no hi ha cap ambigüitat, simplement *vehicles* (fig 4.1).

La derivació de classes aporta una forma de relacionar classes (conceptes) entre elles tot i definint una *relació de particularització*.

> Una classe A és derivada d'una altra classe B si
>
> - A comparteix l'estructura general definida per la classe B i
> - A afegeix o redefineix algun element (atribut o operació) a l'estructura definida per B.

Altres maneres equivalents d'entendre intuitivament el que diu aquesta definició podrien ser les següents:

Una classe A és derivada d'una altra classe B si:

- A és una especialització de B.
- A *és un* B.
- A és una particularització de B.
- B és una generalització de A.

A la classe B se l'anomena *classe base* o *superclasse* i a la classe A, *classe derivada* o *subclasse*.

Si continuem amb l'exemple dels vehicles, podríem considerar els atributs *color, nombre de cilindres i velocitat màxima* com a atributs generals de tots els vehicles. Ara bé, per a un camió, a més a més d'aquests atributs generals caldria definir-ne altres, com ara *la capacitat de càrrega o el nombre d'eixos*, per a una furgoneta seria important la seva *capacitat* i per a un turisme, per exemple, el *número de portes*. La definició d'aquests atributs més específics seria la manera en què els camions, furgonetes i turismes particularitzarien els atributs generals dels *vehicles*.

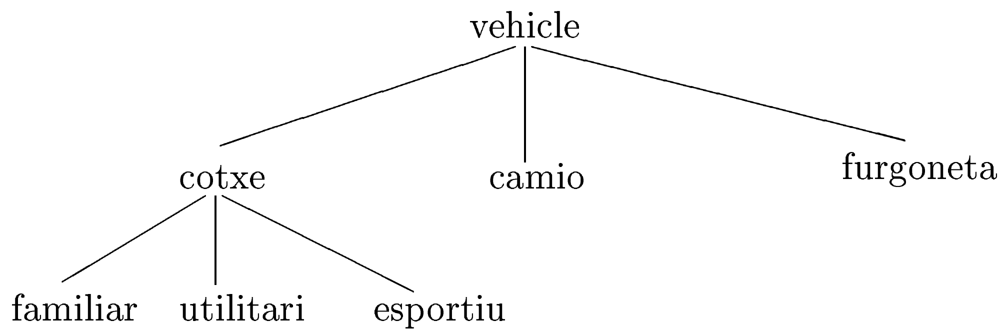


Figura 4.1: Jerarquia de classes pels vehicles

4.2 Herència

Un aspecte important de la definició és que deixa ben clar que la classe derivada *comparteix* l'estructura de la classe base (i possiblement hi afegeix algun element addicional). Podem presentar això d'una altra manera dient que la classe derivada *hereta* l'estructura de la seva superclasse. L'herència és una propietat molt important de la POO i el seu sentit el podríem sintetitzar de la següent manera:

> L'herència és la propietat per la qual una classe derivada conserva la mateixa estructura (atributs i operacions) que la seva classe base [1].

la classe base defineix una interfície que serà compartida però també especialitzada per totes les seves subclasses.

4.2.1 Exemple 1

```
class objecte_casa {

  int preu;
  char lloc[MAXC];
  char nom[MAXC];
  char marca[MAXC];

public:
  objecte_casa(char* plloc, char* pmarca, char* pnom, int ppreu);
  ~objecte_casa();
  void obt\_nom(char* pnom);
  void llista_caract();
  //altres operacions
};



class roba : public objecte_casa{

public:
  roba(char* plloc, char* pmarca, char* pnom, int pr)
```

[1] Amb l'excepció feta dels constructors que no s'hereten. Veure 4.4.

```
      :objecte_casa(pr,mar,llo,no)
      {}
  //altres operacions

};


class aliment : public objecte_casa{

public:
  aliment(char* plloc, char* pmarca, char* pnom, int pr)
      :objecte_casa(pr,mar,llo,no)
      {}
  //altres operacions

};
```

En aquest exemple:

- Declarem les classes *roba* i *aliment* com a subclasses de la classe *objecte_casa*. Això ho fem en $C++$ de la següent manera:

  `class roba:  public objecte_casa`

  El significat del delimitador `:public` el presentem a 4.7.

- Fem que *roba* i *aliment*, per ser subclasses d'*objecte_casa*, hereten els atributs `nom, preu, marca` i `lloc` de *objecte_casa* i també l'operació `llista_caract` (i altres que poguessin estar definides a *objecte_casa*. Els constructors no s'hereten[2].

- Hem omitit el codi de les operacions. Més endavant (4.2.3) el proposarem.

Com a conseqüència de les definicions que hem fet de les classes *roba* i *aliment*, podem fer:

```
aliment a1(çuina", ĿLa Seu", çafe", 300);
a1.llista_caract();
a1.obt_nom(cadena);
```

(...)

Normalment una subclasse obtinguda d'una relació d'especialització no es limitarà a *heretar* l'estructura i el comportament de la superclasse a la que especialitza sinó que els farà més adients a les seves pròpies característiques. Aquesta especialització la pot fer de les següents maneres:

- Afegir operacions a les que hereta de la superclasse.
- Afegir atributs als que hereta de la superclasse.
- Fixar els valors d'algun atribut que ha heretat de la superclasse. Per exemple, si definim *taxi* com a subclasse de *vehicle_de_4_rodes* i ens limitem a la ciutat de Lleida, podrem fixar l'atribut *color* de la subclasse a *blanc*.
- Redefinir operacions de la superclasse (donar un significat més específic a alguna operació de la superclasse).

Presentem tot seguit dos exemples on apareixen aquestes formes d'especialitzar la definició d'una superclasse.

[2] A l'apartat 4.4 es fa una discusió més detallada dels constructors en les classes derivades i es dóna el seu codi.

4.2.2 Exemple 2

Les relacions presentades a l'exemple dels vehicles les podríem fer explícites en C++ de la següent manera:

```
class vehicle {

  int num_cilind;
  char[MAXC] color;
  int veloc_max;

public:

  vehicle (int pn_cil, char* pcolor, int pveloc_max);
  ~vehicle ();
  void llista_caract();
};


class furgoneta :public vehicle {

int capacitat;

//(...)
};

class turisme :public vehicle{

int num_portes;

//(...)

};

class familiar :public turisme{

int capac_viatgers;

//(...)

};
```

A la definició de cada subclasse s'indica:

- Quina és la seva superclasse: `class furgoneta :public vehicle` o bé `class familar :public turisme` (el significat exacte del delimitador `:public` el presentarem a 4.7).
- Quins són els atributs o operacions que la subclasse afegeix a la superclasse.

Notem també que no hem de repetir els atributs que la subclasse comparteix amb la superclasse perquè aquella ja els *hereta* per defecte.

4.2.3 Exemple 3

En aquest exemple enriquim la classe *roba* amb els atributs *material* i *talla* i la classe *aliment* amb els atributs *caducitat* i *envàs.*

```
class objecte_casa {

int preu;
char lloc[MAXC];
char nom[MAXC];
char marca[MAXC];

public:
  objecte_casa(){};
  objecte_casa(char* plloc, char* pmarca, char* pnom, int ppreu)
   {
     strcpy(lloc, plloc);
     strcpy(nom, pnom);
     strcpy(marca, pmarca);
     preu=ppreu;
   }
  int obt_preu(){return preu;}
  void obt_marca (char* pmarca){strcpy(pmarca, marca);}
  void obt_lloc (char* plloc){...}
  void obt_nom (char* pnom){...}
  void llista_caract()
    {
     cout << "nom=" << nom << "lloc=" << lloc;
  << "marca=" << marca;
    }
};


class roba : public objecte_casa{

char material[MAXC];
int talla;

public:

  roba(char* plloc, char* pmarca, char* pnom, int ppreu,
       char* pmat, int ptalla)
    :objecte_casa(plloc, pmarca, pnom, ppreu), talla(tall)
    {
      strcpy(material, pmat)
    }
  int obt_talla() {retorna talla;}
  void obt_material(char* pmat){strcpy(pmat, material);}
  void llista_caract()
    {
      objecte_casa::llista_caract();
      cout <<" material=" << material
```

```
   << "talla=" << talla;
    }
};


class aliment : public objecte_casa{

data caducitat;
char contenidor[MAXCAR];

public:

aliment(char* pmar, char* pnom, int ppreu,
data pcad, char* pconte )
  :objecte_casa("cuina",pmar,pnom,ppreu),caducitat(cad)
  {
    strcpy(contenidor,pconte);
  }
void obt_caduc (data& cad) {cad=caducitat;}
void llista_caract()
  {
    objecte_casa::llista_caract();
    cout << "caducitat="<< caducitat
 << "contenidor=" << contenidor << endl;
     }
};
```

Tot seguit enumerem els aspectes més destacats que són il.lustrats per aquest exemple:

- Les classes *aliment* i *roba* hereten de la classe *objecte_casa*:
  - Els atributs `nom, lloc, preu i marca`.
  - Les operacions `obt_lloc, obt_preu, obt_marca i obt_nom`

  En conseqüència, un objecte de la classe *aliment* disposarà dels atributs `nom, lloc, preu i marca` i podrà cridar les operacions `obt_lloc, obt_nom ....` tot i que no hagin estat definides en aquesta classe.

  Per exemple, serà perfectament lícit fer:

  ```
  aliment a1(...);

  cout << a1.obt_preu();
  ```

- La classe *aliment* afegeix a la superclasse *objecte_casa* els atributs `caducitat` i `contenidor` i les operacions `obt_caduc()` i `obt_contenid()`. Una cosa semblant es pot dir de la classe `roba` amb els atributs *talla* i *material*.
- La classe *aliment* fixa l'atribut `lloc` al valor *cuina*.

- Les classes *aliment* i *roba* redefineixen l'operació `llista_caract`.

  L'operació *llista_caract* definida en la classe *objecte_casa* és heretada per les classes `roba` i `aliment`. Malauradament, el comportament d'aquesta operació no s'adapta completament als nostres desitjos: efectivament, *llista_caract* llistarà els valors dels atributs comuns *marca*, *lloc*, *preu* i *nom* però no els dels atributs *caducitat* i *contenidor*, en el cas dels aliments, o *material* i *talla* per la roba. D'aquesta manera es justifica la necessitat que tenen les classes `roba` i `aliment` de *redefinir* el significat d'aquesta operació per tal d'adaptar-la a les seves necessitats.

  A l'exemple apareix el codi d'aquesta operació redefinida a totes dues subclasses. Ens podem adonar fàcilment que el codi de *llista_caract* a la classe *objecte de la casa* s'encarrega de llistar els valors dels atributs comuns (nom, preu, lloc i marca), mentre que el codi d'aquesta operació a qualsevol de les subclasses, consta de dues parts:

  1. Crida a *llista_caract* definida a la superclasse (per a llistar els atributs comuns) i
  2. Llistat dels atributs específics de la subclasse (*contenidor* i *data de caducitat* per aliment).

- El codi dels constructors de les classes derivades (*roba* i *aliment*) necessiten alguns comentaris addicionals que seran presentats a l'apartat 4.4.

- Els operadors d'entrada/sortida (<<) es descriuen a ???.

Ara, si tenim definits i inicialitzats els següents objectes:

```
objecte_casa o1 ("cuina","coral","rentaplats",200);
aliment a1 ("La Seu","cafe",300, "paquet", d_cadu);
roba r1 ("Ferrys", "samarreta", 1000, "coto", 42);
```

(Suposem que `d_cadu` és un objecte de classe *data*).

Podem fer operacions tals com:

```
o1.llista_caract();

a1.llista_caract();

r1.llista_caract();
```

La primera invocació de l'operació executaria el codi de `objecte_casa::llista_caract()` (i, per tant, llistaria el nom, la marca, el lloc i el preu de l'objecte). La segona invocació executaria el codi de `aliment::llista_caract()` (i llistaria, a més a més dels atributs comuns, el contenidor i la data de caducitat). La tercera, buscaria `llista_caract()` a la classe *roba* i llistaria tots els atributs específics de la classe *roba* a més a més dels comuns.

A més a més d'aquestes dues classes, podem recuperar la classe *medicament* (veure 2.10.3) i la podem veure com a particularització d'*objecte de la casa*, afegint-hi els atributs:

- `metge` que la va receptar i
- `malaltia` que cura.

La nova classe *medicament* definida com a subclasse d'*objecte de la casa* quedaria de la següent manera:

```
class medicament :public objecte_casa{

  char* metge;            //en lloc de char metge[MAX];
  char* malaltia;         //en lloc de char malaltia[MAX];

public:
  medicament(char* plloc, char* pmarca, char* pnom int ppreu,
     char* pmetge, char* pmalaltia);
  ~medicament();
  void llista_caract();

};
```

A l'apartat 4.4 s'explica com manejar els constructors i destructors de les classes derivades. Deixem per aquest apartat la discusió del codi de les operacions constructora i destructora de *medicament, aliment i roba.*

4.3 Subtipus

Fins ara hem presentat l'herència com una propietat que permet enriquir una classe amb nous atributs i funcionalitats: la subclasse conserva l'estructura de la superclasse i hi afegeix noves funcionalitats (o modifica les ja existents).

Les relacions *és_un* que lliguen la subclasse amb la seva superclasse permeten també d'introduir el terme de *subtipus.*

Quan diem que un *cotxe és un vehicle* estem dient que en tots els contextos en els quals es pot parlar de *vehicles* podrem parlar de *cotxes* (encara que a l'inrevés no sigui cert).

Una altra manera de dir això mateix és que la classe *cotxe* (subclasse) implementa un subtipus de la classe *vehicle* (superclasse) o bé que una subclasse té el mateix tipus base que la seva superclasse.

Les conseqüències d'aquest punt de vista en la programació queden paleses en els següents aspectes:

Assignació El llenguatge permet fer l'assignació d'un objecte d'una subclasse a un objecte de la seva superclasse.

```
vehicle v(...);
cotxe c(...);

v=c;    //Correcte.

c=v;    //Error. Assignacio no segura.
```

La lectura que cal fer d'això és la següent:

Com un cotxe és un vehicle, no hi ha cap problema per a assignar un cotxe a un vehicle

Pas de paràmetres El llenguatge permet associar al paràmetre formal (de classe C) d'una acció un paràmetre actual d'una subclasse de C.

```
void f(vehicle& v)
{

//...

}


main()
{

  cotxe c(...);

  f(c);

  //...

}
```

Aquesta possibilitat l'hem de llegir de la següent manera:

Com un cotxe és un vehicle, podem passar a l'acció f un cotxe.

La utilitat de tot això la veurem amb escreix a l'apartat 5.

4.4 Els constructors en les classes derivades

Que *aliment* sigui una classe derivada de *objecte de la casa* vol dir que quan construim una instància de la classe *aliment*, estem també construint una instància de la classe *objecte de la casa* (recordem que *aliment* hereta els atributs d'*objecte de la casa*). Dit d'una altra manera: *el constructor de la classe derivada haurà de cridar al de la classe base i ho haurà de fer amb els arguments que tingui aquest.*

Una classe derivada no hereta mai els constructors de la seva classe base.

4.4.1 Exemple

Recordem el codi del constructor de la classe aliment presentada a 4.2.3:

```
aliment::aliment(char*  plloc,char* pmarca, char* pnom, int ppreu,
 data cad, char* pconte)
    :objecte_casa("cuina",pmar,pnom,ppreu), caducitat(cad)
{
  strcpy(contenidor, pconte);
}
```

Considerem, tot seguit, alguns detalls rellevants d'aquest exemple:

- `:objecte_casa(çuina",pmarca,pnom,ppreu)` no és més que una crida explícita al constructor de la classe base *objecte_casa* (veure 4.2.3).

- `caducitat(cad)` és una crida explícita al constructor copiador de la classe *data* (definit a 2.9).
- Notem que d'aquesta manera estem fixant l'atribut `lloc` amb el valor `cuina` per a tots els aliments.
- Finalment, aquest és un exemple d'una relació d'agregació *aliment-data*. Una relació d'agregació es caracteritza pel fet que una classe (en aquest cas, *aliment*) conté com a atribut un objecte d'una altra classe (en aquest cas *data*).

4.4.2 Exemple 2

Tot seguit presentem el constructor de la classe *medicament* (veure 4.2.3:

```
class medicament :public objecte_casa{

  char* metge;
  char* malaltia;

public:
  medicament(char* plloc, char* pmarca, char* pnom, int ppreu,
     char* pmetge, char* pmalaltia)
       :objecte_casa(plloc,pmarca,pnom,ppreu)

  {
    metge=new char[strlen(pmetge)+1];
    malaltia=new char[strlen(pmalaltia)+1];
    strcpy(metge,pmetge);
    strcpy(malaltia,pmalaltia);
  }

//altres operacions...
};
```

4.5 Els destructors en les classes derivades

L'aspecte més rellevant que afecta els destructors de les classes derivades és que aquests criden implícitament el destructor de la seva classe base.

Aquest fet és necessari si es considera que un objecte de la classe derivada és, en particular, un objecte de la classe base de la qual hereta alguns atributs. Només el destructor de la classe base sap com destruir aquells atributs (definits a la classe base i heretats per la classe derivada).

4.5.1 Un exemple

Considerem les dues classes següents:

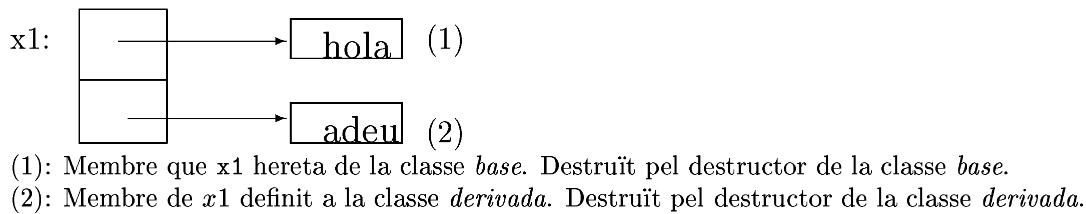
```
class base{

  char* c1;
```

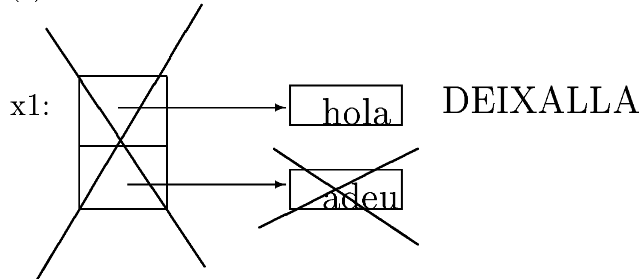
```
public:

 base(char* cadena)
 {
    c1=new char[strlen(cadena)+1];
    strcpy(c1,cadena);
 }
 ~base()
 {
   delete [] c1;
 }
};

class derivada{

 char* c2;

public:

 derivada(char* cadena1, char* cadena2)
    :base(cadena1);
 {
    c2=new char[strlen(cadena2)+1];
    strcpy(c2,cadena2);
 }
 ~derivada()
{
  delete [] c2;
}

};

main()
{
  derivada x1("hola","adeu");
}
```

Al programa principal es crea un objecte anomenat `x1` de la classe *derivada* amb dues cadenes com a paràmetres. Aquestes dues cadenes serviran com a inicialització a l'atribut `c2` definit a *derivada* i a l'atribut `c1` definit a *base* i heretat per *derivada*. Quan acaba l'acció *main* es realitza una crida implícita al destructor de *derivada*, el qual s'encarrega de:

1. Destruir l'atribut `c2` tot i alliberant la memòria dinàmica reservada en la seva construcció.
2. Cridar el destructor de la classe *base* el qual s'encarregarà de destruir l'atribut `c1` tot i alliberant la memòria dinàmica reservada en el moment de la seva creació.

A la figura 4.2 es presenta gràficament el procés de destrucció i també les conseqüències que es produirien si no es fes la crida automàtica al destructor de la classe base.





Destrucció de `x1` si el destructor de la classe base no estés definit.

Figura 4.2: Procés de destrucció de `x1`

4.6 Els tipus de membres

L'encapsulament és una de les propietats caracteritzadores del model de la P.O.O.. *Encapsulament* vol dir l'ocultació de determinats detalls (membres) d'una classe als clients d'aquella classe. D'aquesta manera, una classe queda definida de cara als seus clients com el conjunt de *membres no ocults* (o públics) d'aquella classe amb un determinat comportament perfectament especificat en algun lloc. A aquest conjunt de membres públics d'una classe (majoritàriament operacions) juntament amb la seva especificació els anomenem *la interfície de la classe.*

La forma que té C++ de manejar l'encapsulament és identificant alguns membres de la classe com a *privats* i altres com a *publics.* Els darrers seran els que s'oferiran a les classes clients mentre que els primers només seran visibles dins de la pròpia classe.

La definició de les classes derivades fa aparéixer encara un nou tipus de membres. El raonament és el següent: Si B és una classe derivada d'A, és raonable pensar que B no ha de ser tractada com una *classe client* més. B és una classe molt lligada a A (en general, en serà una especialització) i, per tant, té sentit que A li deixi accedir a alguns dels seus membres que són ocults per a les *classes client* en general. Són els membres *protegits.*

En conclusió, una classe pot tenir tres tipus de membres: els `private`, els `protected` i els `public`.

- Els membres `private` d'una classe només poden ser accedits per:
  - Les funcions membres de la pròpia classe i
  - Les funcions *friend* (veure 7) d'aquella classe.
- Els membres `protected` d'una classe A poden ser accedits per:
  - Les funcions membres de la pròpia classe,
  - Les funcions *friend* d'aquella classe,
  - Les funcions membres de qualsevol classe derivada d'A.
  - Les funcions *friend* de qualsevol classe derivada d'A.
- Els membres `public` d'una classe poden ser accedits per qualsevol funció de qualsevol altra classe.

Veiem un exemple:

```
class A {

private:

  int x;
  void f1();

protected:

  int y;
  void f2();

public:

  int z;
  void f3();

};

class B :public A {

  int t;

public:

  void g();

};


class C {

public:

  void h();
};
```

Veiem ara els permisos d'accés:

- Des dels codis de les funcions *f1(), f2()* i *f3()* es podrà accedir a tots els membres de A, a $g()$ i a $h()$.
- Des del codi de la funció *g()* es podrà accedir a y, z, t, $f2()$ i $f3()$ però no a $f1()$ ni a x.
- Des del codi de la funció *h()* es podrà accedir a z, $g()$ i $f3()$, però no a y, $f2()$, x, $f1()$ ni a t.

4.7 Els tipus d'herència

Siguin A i B dues classes definides de la següent manera:

```
class A {

  int i;

protected:

  int j;

public:

  void f();

};

class B :???? A{

 int k;
public:

 void g();

};
```

La notació `:????` indica que el grup de signes '?' han de ser substituïts per alguna de les paraules `public`, `protected` o `private`[3].

Amb la definició que hem fet, i independentment de `????`, està ben clar que la classe B és una subclasse de la classe A i, en conseqüència, tindrà tres atributs i dues operacions[4]:

- Els atributs `i` i `j` heretats de A.
- L'atribut `k` definit a la mateixa classe B.
- L'operació `f` heretada de A i
- L'operació `g` definida a la mateixa B.

La visibilitat que es tindrà dins la classe B d'aquests elements depén de la definció dels membres (veure 4.6) i és la següent:

Dins la classe B (i.e. en el codi d'alguna acció definida en la classe B) es podran referenciar els membres atribut `j` i `k` i les funcions membres `f` i `g`. En canvi, no es podrà referenciar el membre `i` perquè està definit com a *privat* a la classe A.

> Així doncs, sabem els drets d'accés de la classe B als membres que hereta de A, el que no està definit enlloc és *els drets d'accés de la resta de classes als membres que B hereta de A*.
>
> Dit d'una altra manera, desconeixem *la manera com mostra B a la resta de classes els membres que hereta de A*.

La forma amb què B mostrarà a l'exterior els membres que hereta de A dependrà del tipus d'herència que s'estableixi entre A i B, el qual, a la seva vegada, quedarà fixat per la paraula que sustitueixi a '`:????`' dins la definició de B. Aquesta paraula clau pot ser `:public`, `:protected` o `:private`. Veiem-ho:

[3]De fet, el tipus d'herència dependrà justament de la paraula que s'escolleixi.

[4]Això vol dir que quan creem un objecte de la classe B disposarà automàticament d'aquests tres atributs i dues operacions.

- Si el tipus d'herència és `:public`, B mostrarà a la resta de classes:
  - Com a `private` de B els membres que eren `private` a A.
  - Com a `protected` de B els membres que eren `protected` a B i
  - Com a `public` de B els membres que eren `public` a A.

  O sigui, el tipus d'herència `:public` manté pels membres heretats, el mateix *status* (els mateixos drets d'accés) que tenien a la classe de la qual s'han heretat[5].

- Si el tipus d'herència és `:protected`, B mostrarà a la resta de classes:
  - Com a `private` de B els membres que eren `private` a A.
  - Com a `protected` de B els membres que eren `protected` a B i
  - Com a `protected` de B els membres que eren `public` a A.

  Per tant, només les classes derivades de B tindran accés als membres que B hereta de A i que estan definits a A com a `protected` o `public`.

- Finalment, si el tipus d'herència és `:private`, B mostrarà a la resta de classes:
  - Com a `private` de B els membres que eren `private` a A.
  - Com a `private` de B els membres que eren `protected` a B i
  - Com a `private` de B els membres que eren `public` a A.

  Per tant, tot i que un objecte de la classe B heretarà els membres de A, no es podrà accedir directament a aquells membres des de cap altra classe, perquè B els mostrarà com a `:private` a la resta de classes.

Suposem, per exemple, dues classes addicionals C i D definides com segueix:

```
class C :public B{

  int m;
public:

//...

};

class D {

  int n;

public:

//...
};
```

- Si l'herència de B respecte de A és `:public`:

[5] Però compte, els membres privats de A no seran visibles dins de les operacions de la classe B.

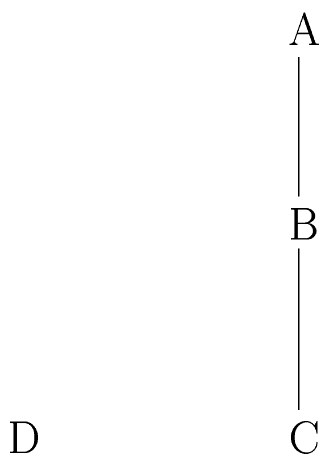


Figura 4.3: Tipus d'herència

  - Des del codi de les funcions definides dins de la classe C es podrà accedir directament als membres j i $f()$ de A heretats per B.
  - Des del codi de les funcions definides dins de la classe D es podrà accedir directament únicament al membre $f()$ que B hereta de A, ja que el membre j, B l'exporta com a `protected` i D no és una subclasse de B.

- Si l'herència de B respecte de A és `:protected`:
  - Dins de la classe C es podrà accedir directament als membres j i $f()$ de A heretats per B, perquè ambdós són exportats per B com a `protected` i C és una subclasse de B.
  - Dins de la classe D **no** es podrà accedir a cap membre de B heretat de A perquè D no és una subclasse de B.

- Si l'herència de B respecte de A és `:private`:

  Ni dins de la clase C ni dins de la classe D no es podrà accedir a cap membre que B hereti de A, perquè aquests membres, B els exporta com a `private`.

El tipus d'herència `:public`, és el més utilitzat en la majoria de les aplicacions que involucren herència perquè es mantenen els drets d'accés definits a la classe original. En particular, si es vol que la classe B, no només hereti l'operació `f()` (definida `public` a A), sinó que també l'ofereixi com a seva, ha d'utilitzar una herència de tipus `:public`.

El tipus d'herència `:private` és també utiltizat en algunes aplicacions en que es desitja que les operacions que una classe hereta no siguin exportades als usuaris d'aquesta classe.

Capítol 5

El polimorfisme

L'herència és en ella mateixa una propietat molt interessant, però ho és més encara quan considerem un concepte que s'aprofita de l'herència i que dóna una potència molt important a la programació orientada a objectes: *el polimorfisme*

Plantegem, en primer lloc, un exemple que justificarà la conveniència del polimorfisme:

5.1 Exemple 1

Considerem la classe *pila d'objectes de la casa*. És clar que un objecte d'aquesta classe podrà contenir, en general, una barreja d'objectes de la classe *aliment* i de la classe *roba*.

Podríem fer una definició d'aquesta classe de la següent manera:

```
class pila_obj_casa {

//Representaci'o escollida

public:

pila_obj_casa();

void empilar(obj_casa&);

void desempilar();
objecte_casa* cim();        //Retorna un referent a l'objecte del
                            //cim de la pila.
bool p_buida();             //Retorna CERT si la pila esta buida i FALS
                            //altrament.
}
```

Estem interessats en escriure una acció *llistar*, client de la classe anterior que llisti les característiques de tots els objectes que formen part d'una determinada pila d'objectes de la casa. A la figura 5.1 proposem un codi per aquesta acció.

El comportament que esperem d'aquesta acció és que llisti totes les característiques de tots els objectes de la pila. Per exemple, si la pila *pobj* està formada per una ampolla de vi i uns texans, *llistar* hauria de

```
void llistar(pila_obj_casa& pobj)
{
   objecte_casa* roc;

   while(!pobj.p_buida()) {
     roc=pobj.cim();
     roc->llista_caract();
     pobj.desempilar();
   }
}
```

Figura 5.1: Una operació per a llistar tots els elements d'una pila

produir:

```
nom: vi
marca: priorat
preu: 300
lloc: rebost
envas: ampolla
caducitat: 1-1-99

nom: texans
marca: Luis
preu: 10000
lloc: armari
talla: 42
material: coto
```

Malauradament, com ja haureu endevinat, l'acció `llistar` no ofereix, ni de bon tros, aquest comportament, sinó més aviat, aquest altre:

```
nom: vi
marca: priorat
preu: 300
lloc: rebost

nom: texans
marca: Luis
preu: 10000
lloc: armari
```

El motiu és que la decisió del codi de les accions associat a cada crida es pren en temps de compilació (el que s'anomena lligam estàtic). Quan el compilador processa la instrucció

`objecte_casa* roc;`

pren nota que, a partir de llavors, haurà de buscar el codi de les operacions que s'apliquin sobre l'objecte $*roc$ a la classe *objecte_casa*, que és on estaran definides. Això vol dir, en particular, que la crida

`roc->llista_caract();`

es lligarà amb l'operació `objecte_casa::llista_caract();`

independentment del tipus real que en temps d'execució tingui l'objecte `*roc`.

D'igual manera, si definíssim `aliment* rali;`, lligaria la crida `rali->llista_caract()` amb l'operació `aliment::llista_caract()`.

Per a resoldre aquest problema podríem pensar en afegir un nou membre a la classe objecte_casa: *tipus*.

```
class objecte_casa {

protected:
 char tipus;

//Resta representaci'o

public:
  objecte_casa(int,char*, char*, char* ){... tipus='o';...}
  ~objecte_casa();
  char obt_tipus(){return tipus;}
  void llista_caract();
  };


class roba {

//Representaci'o...

public:
  objecte_casa(int,char*, char*, char* ){... tipus='r';...}
  ~objecte_casa();
  char obt_tipus(){return tipus;}
  void llista_caract();
  };
```

I ara, l'acció *llistar* quedaria de la següent manera:

```
void llistar2 (pila_oc& pobj)
{
   objecte_casa* roc;

    while(!pobj.p_buida()) {
        roc=pobj.cim();
        switch (roc->obt_tipus()) {
          case 'a':
            ((aliment*)(roc))->aliment::llista_caract();
            break;
          case 'r':
```

```
            ((roba*)(roc))->roba::llista_caract();
            break;
          case 'o':
             roc->llista_caract();
             break;
          default:
          cout<<"error\n";
       }

       pobj.desempilar();
   }

}
```

Si bé aquesta solució funciona i obté el resultat esperat, haurem d'estar d'acord en almenys tres qüestions:

1. El codi és molt poc elegant. Ja ho és ara, però podríem empitjorar-ho si consideréssim, no dues, sinó moltes més classes derivades (com ara *moble, medicament, detergent, sabata....*). Tampoc contribueixen gaire a l'elegància les conversions explícites de tipus tals com `((roba*)(roc))`, necessàries per a cridar a `roba::llista_caract()`.

   Fóra possible conservar l'elegància de llistar però guanyar la funcionalitat desitjada?

2. El programa és fràgil doncs depén de que el programador es recordi de fer la comprovació de tipus. Això que pot semblar trivial i evident en un programa petit, no ho és tant, en programes molt més llargs i complexos. Seria molt millor que la comprovació de tipus es fes de manera automàtica.

3. Comprometem la reutilització del codi. Aquest és un dels pilars de la POO i, sorgeix com a requeriment fonamental de l'enginyeria del software, en la qual els programes estan sotmesos a una evolució continuada des del seu disseny inicial per adaptar-los a noves funcionalitats que en un moment determinat es revelen com a necessàries. En contrast, l'acció *llistar2* no ho és gens, de reutilitzable. Quan s'hagin de crear i utilitzar noves subclasses de *objecte_casa* (per exemple, *detergent*), caldrà modificar el seu codi per a permetre el llistat de les característiques dels detergents i, per suposat, recompilar-lo.

   Fóra possible, afegir noves subclasses de manera que els vells programes que utilitzaven la superclasse no se'n vegin alterats?.

La resposta a ambdues qüestions és *sí*. Potser un *sí* amb alguna matització...

Com ja hem introduït, la dificultat amb l'aproximació proposada sorgeix del requeriment que tenen els llenguatges de programació tradicionals en el sentit que quan comença a executar-se un programa, *totes les crides a funcions i accions han d'estar ja lligades amb el seu codi* (lligam estàtic o *early binding*). D'aquesta manera, l'acció *llistar* té associat de forma estàtica el codi de `objecte_casa::llista_caract` a la crida `llista_caract` , sense saber en realitat sobre quina classe d'objectes s'haurà d'executar l'acció (roba o aliment).

De la mateixa manera, *llistar2* té associat estàticament un codi a cadascuna de les crides a `llista_caract`.

Amb el plantejament que hem fet, no és difícil d'imaginar que una proposta de solució podria consistir en estalviar el requeriment de que el lligam entre la crida a la funció i el seu codi hagi d'estar fet en temps de compilació. Podríem fer aquest lligam en el moment de l'execució de la crida, depenent de l'objecte amb el qual es fa la mateixa. Un tipus de lligam entre les crides i els codis de les accions que compleixi aquest requeriment s'anomena *lligam dinàmic o late binding* i és preceptiu en tots els llenguatges de programació que vulguin tenir el qualificatiu de ser *orientats a objectes.*

El que estem dient és que la mateixa crida:

`roc->llista_caract()`

pugui tenir comportaments diferents segons la classe real a la que pertanyi l'objecte **roc* (la qual cosa es pot saber només en temps d'execució):

- Si **roc* és de la classe *aliment*, `roc->llista_caract()`, executarà el codi de `aliment::llista_caract()`.
- Si **roc* és de la classe *roba*, `roc->llista_caract()`, executarà el codi de `roba::llista_caract()`.

Amb el lligam dinàmic el codi de la figura 5.1 té el comportament desitjat.

D'aquesta propietat se'n diu *polimorfisme*.

> Anomenem *polimorfisme* a la capacitat que té una mateixa crida a acció de presentar diferents comportaments en moments diferents (i.e. lligar-se amb diferents codis en diferents execucions de la mateixa crida).

Intuïtivament, el mètode triat per a executar és el que correspon a la classe més específica a la que pertany l'objecte sobre el qual es fa la crida.

La decisió de quin és el mètode que cal executar en la crida a un mètode polimòrfic es pren en temps d'execució perquè fins llavors no se sap quina serà la classe (tipus) més específica de l'objecte sobre el qual es fa la crida.

D'aquesta darrera consideració se n'extreu una conseqüència important:

> Quan dotem al nostre model de programació de polimorfisme lligat a les relacions d'herència, el lligam entre un objecte que apareix al programa (en realitat, una variable del programa) i el seu tipus s'ha de fer en temps d'execució. Recordem que fins ara, en temps de compilació era sempre possible determinar el tipus d'una variable.

Un concepte que pot confondre's amb el de polimorfisme és el de *sobrecàrrega*. Una acció o funció està sobrecarregada si n'existeix alguna(es) altra(es) amb el mateix nom però diferent nombre o tipus de paràmetres.

La sobrecàrrega la trobem en un ventall prou important de situacions diferents:

1. Sobrecàrrega d'operadors i funcions:

   És el cas dels operadors aritmètics:

   +(enter, enter) $\longrightarrow$ enter
   +(real,real) $\longrightarrow$ real
   +(complex, complex) $\longrightarrow$ complex

2. Classes genèriques:

   push(pila_enters, enter) $\longrightarrow$ pila_enters
   push(pila_caracters,caracter) $\longrightarrow$ pila_caracters

3. Jerarquies de classes:

   obtenir_preu(objecte_casa)$\longrightarrow$ enter
   obtenir_preu(roba) $\longrightarrow$ enter

Una acció polimòrfica és una acció sobrecarregada dins d'una jerarquia de classes tal que el lligam entre la crida i el codi es fa en temps d'execució. Com veurem més avall, en C++ les úniques accions polimòrfiques són les definides com a `virtual`.

Presentem, per acabar, els avantatges del polimorfisme (tal i com es posen de manifest al codi de la figura 5.1), si és que no són prou evidents:

1. El codi de l'acció *llistar* és independent del tipus d'objectes que conté la pila (sempre i quan siguin objectes de la casa). Això vol dir que l'acció *llistar* llistarà correctament tots els atributs de tots els elements de la pila independentment de si són peces de roba o aliments[1]. El polimorfisme dóna independència al codi client respecte certes modificacions de la jerarquia de les classes que utilitza.

2. Si en un futur es defineixen nous tipus d'`objectes_casa` com ara *electrodomèstics*, la mateixa acció *llistar*, sense modificar el seu codi, ens permetrà llistar electrodomèstics amb tots els seus atributs. Només caldrà definir aquesta nova subclasse d'`objectes_casa` i dotar de significat els seus mètodes.

   Podrem, doncs, afegir classes d'objectes sense haver de modificar el codi de les accions client. La propietat del polimorfisme dóna reusabilitat al codi.

Fins aquí hem presentat algunes generalitats sobre el polimorfisme. Ara ens cal veure de quina manera el llenguatge de programació C++ permet treballar amb polimorfisme i amb lligams dinàmics. Les eines que defineix C++ per a fer-ho són les *funcions virtuals* i les *classes abstractes.*

5.2 Les funcions virtuals

Ja hem dit que la característica que ha de tenir un llenguatge de programació per a poder tractar amb el polimorfisme és que ha de permetre els lligams dinàmics (late binding) entre una crida a acció i el codi que s'executarà de la mateixa. C++ té aquesta característica... però no pas per a totes les funcions. La implementació dels lligams dinàmics té un cert cost i això fa que, per a optimitzar recursos, C++ només permeti l'aplicació d'aquest tipus de lligams a *aquelles operacions d'una classe que han estat definides com a virtuals*[2].

> Una acció (operació) virtual és una operació d'una classe que es lliga a una crida a acció en el moment d'executar-se aquella crida i no en temps de compilació com passa amb la resta de les accions (lligam dinàmic).
>
> El lligam dinàmic només s'aplica a les crides a operacions virtuals fetes sobre un apuntador o una referència a un objecte[3].
>
> El lligam entre crida i acció virtual en temps d'execució es fa segons el tipus més específic de l'objecte sobre el qual es fa la crida (i no segons el tipus assignat a la definició de l'objecte).
>
> En general, si una operació d'una classe es defineix com a virtual, serà redefinida per alguna de les seves subclasses.

Tot seguit posem l'exemple:

5.2.1 Exemple 1

Recordem l'exemple amb què vam presentar el polimorfisme a 5.1. Es tractava de suposar que disposàvem d'una classe *pila d'objectes de la casa* amb les operacions habituals de les piles. Volíem dissenyar una acció client d'aquesta classe anomenada `llistar` que havia de tenir com a objectiu presentar per la sortida estàndar els objectes (de la casa) que contenia aquesta pila[4].

L'acció `llistar` que proposàvem és la de la figura 5.1.

[1]Quan l'acció *llistar* fa la crida `roc->.llista_caract` no és l'acció *llistar* la que decideix quin mètode concret *llista_caract* cal aplicar sinó que és el propi **roc* qui ho decideix segons el tipus d'aquest objecte sigui *aliment* o *roba.*

[2]En conseqüència, les úniques funcions que poden ser virtuals són les que han estat definides com a membres d'una classe.

[3]A l'apartat 5.2.3 ampliem això.

[4]L'exemple de la pila d'objectes de la casa el desenvoluparem detalladament a l'apartat 5.6.

Ja hem vist que si el lligam entre la crida `llista_caract()` i el seu codi és estàtic, `llistar` funcionarà malament perquè la crida `roc->llista_caract()` s'associarà en temps de compilació amb l'acció `objecte_casa::llista_caract()` independentment del tipus real que, en temps d'execució, tindran els diferents objectes de la pila. Com a conseqüència, pels objectes de la pila que siguin aliments o peces de roba mai no es llistarà la seva data de caducitat, tipus d'envàs, talla o material.

La solució passarà per aconseguir que el lligam entre la crida a `llista_caract()` i el seu codi sigui dinàmic. Per a que aquest lligam pugui ser dinàmic haurem de fer *virtuals* les accions `llista_caract()`:

```
class objecte_casa {

int preu;
char lloc[MAXC];
char nom[MAXC];
char marca[MAXC];

public:
  objecte_casa(){};
  objecte_casa(int,char*, char* ,char*);
  virtual void llista_caract();
};


void objecte_casa::llista_caract()
  {cout << "nom=" << nom<<"\n";
   cout << "lloc=" << lloc<<"\n";
   cout << "marca=" << marca <<"\n";
   cout << "preu=" << preu <<"\n";
  };
```

La declaració com a *virtual* de l'operació `llista_caract` s'estén de forma automàtica a totes les operacions `llista_caract` que apareguin en les subclasses d'*objecte_casa*. Per tant, a les subclasses *roba i aliment* no és necessari definir `llista_caract` com a virtual. De tota manera, ho farem per raons d'elegància i legibilitat:

```
class roba : public objecte_casa{

char material[MAXC];
int talla;

public:

  roba(char*, char*, char*,int, char*, int);

  int obt_talla();

  void obt_material(char*);
```

```
  virtual void llista_caract() {
     objecte_casa::llista_caract();
     cout<<"talla: "<<talla<<"\n";
     cout<<"material"<<material<<"\n";
  }
};


class aliment : public objecte_casa{

data caducitat;
char envas[MAXCAR];

public:

 aliment(char*, char*, int, data, char*);

 void obt_caduc (data& cad);

 virtual void llista_caract()
 {
  objecte_casa::llista_caract();
  cout<<"data caducitat: ";
  caducitat.llistar_data();  // Sup que la classe data tingui aquesta
                             // accio definida.
  cout<<"tipus envas: "<< envas<<"\n";
 }
};
```

Amb la definició de *llista_caract* com a *acció virtual*, l'acci'o *llistar* ja funcionarà com esperem: Quan l'objecte llegit de la pila sigui un aliment, n'escriurà la data de caducitat i el tipus de contenidor com a característiques específiques. Quan sigui una peça de roba, n'escriurà la talla i el material. Per tots els objectes escriurà les característiques comunes, com ara el nom, la marca o el preu.

A la figura 5.2 es presenta gràficament aquest procés.

Quan vulguem incorporar nous tipus d'objectes de la casa (com ara *medicament*), l'única cosa que ens caldrà fer és dotar-los d'una operació `llista_caract` que cridi a `llista_caract` de `objecte_casa` per a escriure les característiques comunes (nom, marca, preu...) i després escriure les específiques de la classe que vulguem incorporar (en el cas del medicament, seran, quina malaltia cura i quin metge l'ha receptada).

```
class medicament{

 //Representacio...
public:
 //altres operacions

 llista_caract(){
  objecte_casa::llista_caract();
  cout<<"Receptada per " <<metge<<"\n";
  cout<<"cura la malaltia "<<malaltia<<"\n";
```

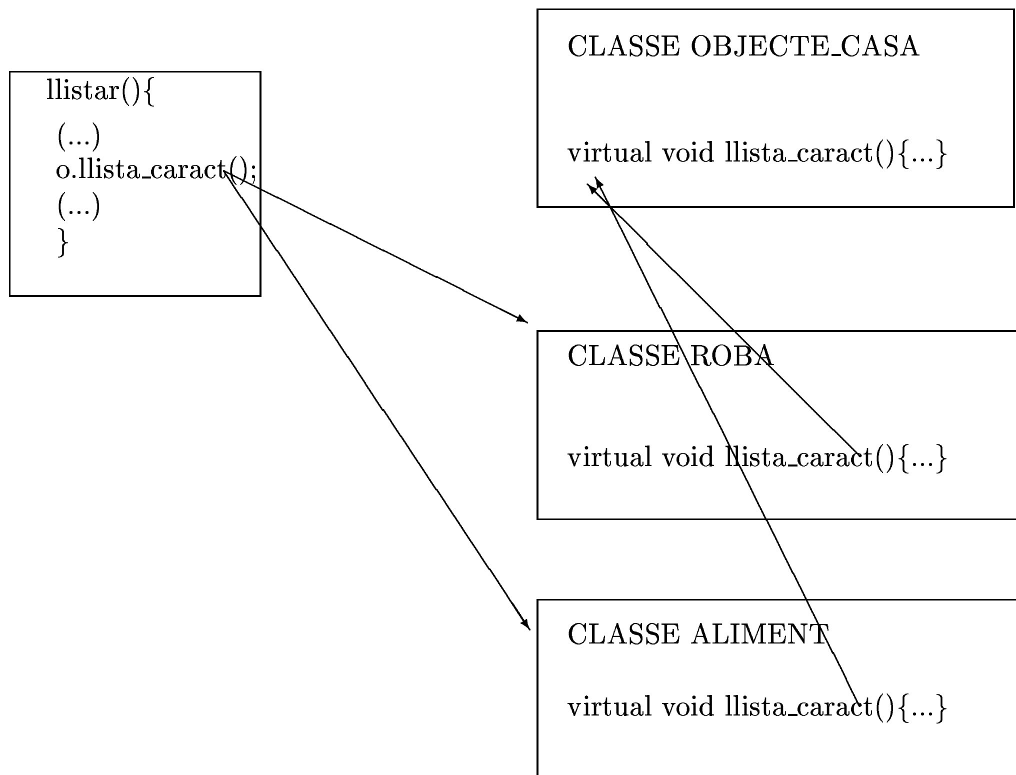


Figura 5.2: Funcionament accions virtuals(1)

```
 }
};
```

L'acció `llistar` seguiria funcionant perfectament, escrivint **totes** les característiques de tots els objectes de la pila fins i tot si hi ha medicaments a la pila.

És curiós adonar-se que l'acció `llistar` funciona correctament amb medicaments tot i haver estat definida *abans* que la classe medicament!!!

Aquesta darrera idea és molt important. És la que fonamenta el tema de la reutilització del codi, aspecte cabdal en la enginyeria del software, com ja hem dit.

5.2.2 Exemple 2

Podem donar un altre exemple que il.lustra el mateix concepte des d'un punt de vista lleugerament diferent: en aquest cas l'acció virtual no es cridarà des de fora de la classe sinó que serà una funció de la pròpia classe la que contindrà una crida a l'acció virtual[5]. Veiem-ho:

```
class objecte_casa {

//Representaci'o

public:
```

[5] A 8.5 veiem una utilitat d'això

```
  //Resta d'operacions

  virtual void llista_caract_espec(){}

  void llista_caract()
     {cout << "nom=" << nom << "lloc=" << lloc
           << "marca=" << marca;
      llista_caract_espec();
     }
  };


class aliment : public objecte_casa{

 data caducitat;
 char contenidor[MAXCAR];

public:

 //Resta d'operacions

 virtual void llista_caract_espec() {
   cout << "caducitat=";
   caducitat.veure_data();
   cout << "contenidor=" << contenidor;
 }
};
```

A la figura 5.3 s'il.lustra la diferència entre aquesta aproximació i la que hem pres a l'exemple anterior.

A l'exemple anterior, l'acció polimòrfica era `llista_caract()` (es cridava a una `llista_caract()` diferent segons el tipus de l'objecte sobre el que es fes la crida).

Ara, l'acció polimòrfica és `llista_caract_espec()`. L'acció `llistar` cridarà sempre la mateixa (i única) `llista_caract()`. Serà aquesta acció la que cridarà a `llista_caract_espec()` de *roba*, d'*aliment* o de *medicament* segons el tipus específic de l'objecte amb el qual s'ha fet la crida.

En conseqüència, no cal definir `llista_caract()` com a virtual, però sí cal donar aquesta propietat a `llista_caract_espec()`.

5.2.3 La crida a les funcions virtuals

Com ja hem dit, per a activar el mecanisme de C++ que maneja el polimorfisme, cal cridar les funcions virtuals amb un *apuntador* o bé amb una *referència*.

Tot seguit analitzem els casos que es poden presentar:

Primer cas: Crida a una funció virtual amb objectes de diferent tipus:

```
main()
{
 objecte_casa oc(...);
```

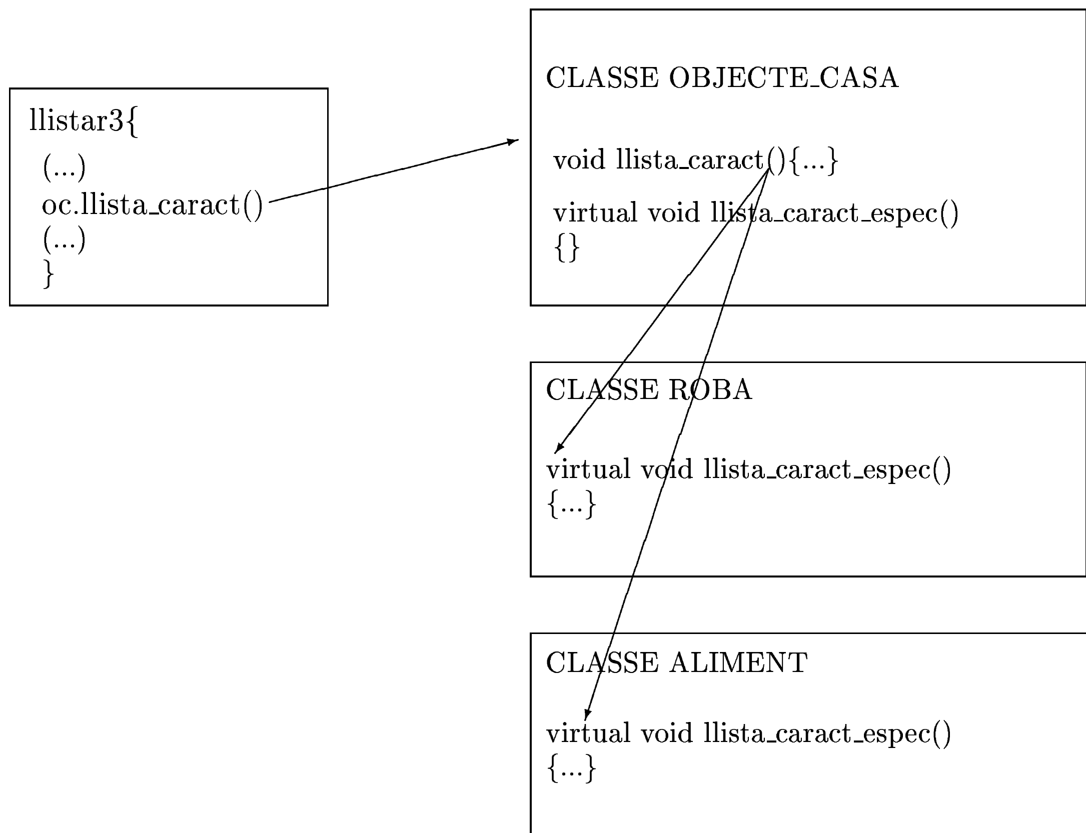


Figura 5.3: Funcionament accions virtuals (2)

```
 aliment al(...);

 oc.llista_caract();
 al.llista_caract();

 //Mes coses...
}
```

Es tracta de crides estàtiques (no polimòrfiques) sobrecarregades (veure 4) que es resolen en temps de compilació. `oc.llista_caract()` cridarà a `objecte_casa::llista_caract()` mentre que `al.llista_caract()` cridarà a `aliment::llista_caract()`. El mecanisme del polimorfisme basat en el lligam dinàmic no s'invoca.

Segon cas: Crida a una funció virtual a través d'un paràmetre passat per valor.

```
void f(objecte_casa x)
{
  x.llista_caract();

}


main()
{
 objecte_casa oc(...);
 aliment al(...);

 f(oc);
 f(al);

 //Mes coses...
}
```

Novament es generarà una crida estàtica (no polimòrfica) que serà també resolta en temps de compilació. La crida `llista_caract()` invocarà la funció que correspongui depenent del tipus estàtic del paràmetre formal que sempre serà *objecte_casa*. Per tant, la crida `x.llista_caract()` sempre es referirà a `objecte_casa::llista_caract()`. Com a conseqüència, el mecanisme del polimorfisme no serà invocat i només es llistaran els atributs comuns `nom`, `marca`, `preu` i `lloc`, tant per `oc` com per `al`.

Tercer cas: Crida a una funció virtual a través d'un paràmetre passat per referència.

```
void f(objecte_casa& x)
{
  x.llista_caract();

}


main()
```

```
{
 objecte_casa oc(...);
 aliment al(...);

 f(oc);
 f(al);

 //Mes coses...
}
```

En aquest cas sí que s'activa el mecanisme del polimorfisme implementat per C++ mitjançant les funcions virtuals. Mentre que un paràmetre passat per valor és un objecte d'un tipus fixat en temps de compilació, un paràmetre passat per referència és internament un apuntador a un objecte (A l'apèndix A s'expliquen les referències amb més detall). En aquestes condicions, si es crida a una funció virtual amb un paràmetre passat per referència, es fa un lligam dinàmic amb l'operació de la classe de l'objecte al que apunta el paràmetre en el moment de la crida.

A l'exemple, `x.llista_caract()` cridarà en primer lloc a `objecte_casa::llista_caract()` i, en segon lloc, a `aliment::llista_caract()`.

Quart cas: Crida a una funció virtual a través d'un paràmetre *apuntador* (referència explícita).

```
void f(objecte_casa* x)
{
  x->llista_caract();

}


main()
{
 objecte_casa oc(...);
 aliment al(...);

 f(&oc);
 f(&al);

 //Mes coses...
}
```

Estem en el mateix cas anterior, la diferència és, bàsicament, sintàctica. Aquí, els apuntadors els tractem explícitament com a objectes referents, mentre que en el cas anterior conservàvem la sintaxi utilitzada per a manejar objectes[6]. En tot cas, igual que abans, es fa la crida a la funció de la classe a la que pertany l'objecte al que es refereix l'apuntador.

Els resultats seran idèntics als del cas 3.

[6]Podríem dir que les referències són apuntadors implícits.

5.3 Les classes abstractes

La forma habitual de treballar amb polimorfisme consisteix en definir una classe *general*, dotada d'operacions virtuals i amb tota una col.lecció de subclasses més específiques, les quals redefineixen el sentit d'aquelles operacions (aquest ha estat el mode de treball que hem seguit amb els objectes de la casa). A vegades la classe general ho és tant que no té sentit associar codi a les operacions virtuals que defineix ni tampoc crear-ne instàncies. En aquests casos és diu que la classe és abstracta. Un exemple podria ser la mateixa classe *objecte_casa*. Si en una determinada aplicació volem treballar només amb tipus específics d'objectes de la casa (com ara medicaments, aliments o peces de roba), segurament no tindrà sentit, per aquella aplicació, definir instàncies de la classe *objecte_casa*, que resultarien massa generals. En conseqüència, només les subclasses d'*objecte_casa* tindran instàncies definides.

Evidentment aquesta decisió depén del domini del problema. Si no vulguéssim fer diferenciacions entre tipus d'objectes de la casa (perquè pel nostre problema només són rellevants les diferències entre animals, persones, vegetals, objectes naturals i objectes de la casa), segurament fóra adient considerar una única classe *objecte_casa* sense cap més subclassificació posterior i amb la possibilitat (òbvia en aquest cas) de definir-ne instàncies.

Un altre exemple d'una classe d'aquestes característiques pot ser la classe *estructura de dades*.

Una estructura de dades és un magatzem que pot contenir una col.lecció de dades d'un cert tipus (classe) component T. Totes les estructures de dades tindran operacions per a afegir un element a la col.lecció d'elements de l'estructura de dades, per a treure'n o per a consultar-ne un de determinat. L'aspecte que diferenciarà una estructura de dades d'una altra serà precisament la forma en què s'estructuren els elements dins de la mateixa (es poden estructurar en forma de pila, llista, cua, arbre...). Cada forma d'estructuració donarà lloc a una estructura de dades diferent.

Considerem la classe general *estructura de dades* amb elements de tipus component T[7]. Aquesta classe podria estar definida de forma molt abstracta de la següent manera:

```
template<class T>
class estructura_de_dades {

int num_elements;          //Nombre d'elements que cont'e
                           //actualment l'e.d.

int num_elements_maxim;    //Nombre m'axim d'elements que pot
                           //contenir l'e.d.

//Altres coses....

public:

estructura_de_dades(int tam_max){num_elements_maxim=tam_max;
 num_elements=0;}

virtual void afegir_element(T&);

virtual void eliminar_element();

virtual T* consultar_element();
```

[7]En C++, la definició d'una classe dependent d'un tipus component(classe genèrica) es fa a través de l'eina anomenada *template* que s'explica al capítol 6.

```
int num_elems_actuals() {return num_elements;}

int capacitat_maxima() {return num_elements_maxim;}

};
```

Fem una sèrie de consideracions sobre aquesta classe:

- No té cap mena de sentit definir instàncies d'aquesta classe. És massa general. Cada estructura de dades particular (pila, cua, cua amb prioritat, arbre...) es comporta d'una manera diferent. Si definíssim una instància d'aquesta classe, no sabríem de quina forma s'ha de comportar!!!. Per això queda clar que l'únic sentit d'*estructura de dades* és com a classe base de les estructures de dades més específiques que definirem com a subclasses d'aquesta.
- Les operacions que proposem són suficientment generals i es poden trobar en gran part de les estructures de dades usuals (piles, cues, arbres, llistes,...) amb comportaments específics i ben definits per cada cas. Per aquest motiu, caldrà redefinir-les en les diferents subclasses de *estructura de dades* que es vagin afegint.
- Per les raons exposades en els punts anteriors, és impossible, dins de la classe *estructura_de_dades*, donar un codi per aquelles operacions. Quan estem en un cas com aquest, definirem les operacions de la següent forma:

  ```
  virtual void afegir_element(T&)=0;

  virtual void eliminar_element()=0;

  virtual T* consultar_element()=0;
  ```

  Les operacions així definides s'anomenen *operacions virtuals pures* i les classes que contenen operacions virtuals pures s'anomenen *classes abstractes.*

  Una classe abstracta és una classe que conté alguna operació virtual pura.

  Les classes abstractes corresponen a classes molt generals, de les quals no té sentit definir-ne instàncies. Les classes abstractes serviran de classe base per a altres de més específiques i que contenen algunes operacions amb un comportament que no es pot definir en la classe abstracta.

  Les subclasses de les classes abstractes d'encarregaran de redefinir les operacions virtuals pures.

No cal que totes les operacions d'una classe abstracta siguin virtuals pures. En realitat, sovint interessarà que hi hagi operacions que no ho siguin. Donem-ne dos motius:

1. Si la classe abstracta té algun atribut privat (com és habitualment el cas), l'única manera d'accedir a aquell atribut és a través d'una operació amb codi (no virtual pura) definida a la pròpia classe abstracta. Un exemple d'això són les operacions de la classe *estructura de dades núm_elements_actuals* i *capacitat_max.* Aquestes dues operacions aporten **l'única** forma d'accedir als atributs *num_elements* i *num_elements_max.*
2. El constructor d'una classe no pot ser mai virtual. Aquesta idea la desenvoluparem en els següents apartats.

5.4 Constructors i destructors virtuals

5.4.1 Constructors virtuals

Comencem repetint que els constructors d'una classe *no poden* ser virtuals. Aquesta declaració de principis ens porta ràpidament a una pregunta: *necessitarem en algun moment construir un objecte del qual coneguem el tipus només en temps d'execució?*

Per a intentar contestar a aquesta assenyada pregunta, val la pena Recordar que els constructors es criden implícitament en la definició d'un objecte i també, que és possible definir un objecte de dues maneres (2.10.1):

De forma estàtica que consisteix en associar un tipus al nom d'una variable i crear un objecte d'aquell tipus que, des d'ara, estarà lligat a aquella variable. Per exemple, `objecte_casa o1(...);`. Aquesta és la forma més habitual de definir objectes i, evidentment, es pot resoldre en temps de compilació i, per tant, no hi ha cap necessitat d'un constructor virtual.

De forma dinàmica que consisteix en crear dinànimicament un objecte anònim i associar-lo a un objecte referent ja existent que *l'apuntarà* . Per exemple:

```
objecte_casa* pob;

//...

pob=new objecte_casa(...);

//...
```

En aquest cas `pob` és l'objecte referent que apunta a l'objecte anònim creat amb la instrucció `new`.

En aquest segon cas, sí que es pot plantejar la necessitat d'un constructor virtual.

Pot passar, efectivament, que tinguem definida una referència (apuntador) a una classe base i vulguem assignar-li un **nou** objecte d'una seva classe derivada que només en temps d'execució podem precisar (exemple: volem fer que `pob` ens apunti a un nou aliment). L'operador `new` ens planteja un problema en aquest cas: necessita conèixer en temps de compilació exactament el tipus que ha de crear (i.e. `new aliment(...)` o bé `new roba(...)`). A l'apartat 5.6 desenvolupem un exemple complet en el qual definim i utilitzem uns nous operadors de *clonació i còpia* per a resoldre el problema de cridar *constructors virtuals* en la creació d'objectes dinàmics.

5.5 Destructors virtuals

Contràriament als constructors, els destructors sí que poden ser virtuals. D'aquesta manera, podrem destruir objectes segons el tipus que tinguin *en temps d'execució.*

A l'apartat 5.6 desenvolupem un exemple complet de treball amb polimorfisme que, en particular, utilitza destructors virtuals.

5.6 Exemple: La pila d'objectes de la casa

Fins ara hem suposat l'existència d'una classe *pila d'objectes de la casa* i, fins i tot, hem gosat escriure el codi d'una acció client que tenia l'objectiu d'escriure tots els components de la pila (*llistar*. Veure la figura 5.1). Així i tot, el desenvolupament del codi d'aquesta classe no és pas trivial. És per això que cloem els apartats dedicats a l'herència i al polimorfisme desenvolupant amb tot detall el disseny de la classe *pila d'objectes de la casa*. Aquest disseny ens servirà per a veure aplicat en un mateix exemple gran part de les eines que hem presentat fins ara.

5.6.1 Primera aproximació

Considerem la classe *pila_objectes_casa*. Aquesta classe haurà de ser capaç de contenir objectes de la casa, tant si aquests són aliments, com peces de roba, com altres subclasses que puguem definir més endavant, o una barreja de totes elles.

Proposem-ne una implementació molt senzilla:

```
class pila_oc{

objecte_casa* t[MAX];
int top;

public:

pila_oc(){top=0;}

void empilar(objecte_casa& x)
{

  t[top]=new objecte_casa;                    //malament!!!
  *(t[top])=x;
  top++;
}

//DESEMPILAR, CIM i altres accions de les piles.
};

}
```

L'acció `empilar` és la que ens interessa. L'objectiu d'aquesta acció és el de crear una còpia del paràmetre x i inserir-lo al vector que s'utilitza per a implementar la pila. El problema és que *empilar* no sap el tipus exacte del paràmetre x fins el moment de l'execució. Aquest tipus podria ser igualment *objecte_casa* , *aliment*, o *roba*. Independentment de quin fos aquest tipus, la instrucció `new` que es proposa sempre reservaria espai per a un nou `objecte_casa` (encara que vulguéssim empilar un *aliment* o una peça de roba). L'operador `new`, doncs, no ens serveix.

5.6.2 La clonació

En definitiva, el problema que se'ns planteja és el de crear dinàmicament instàncies de classes que només coneixem exactament en temps d'execució i inicialitzar-les a un determinat valor. Per a fer això no podem utilitzar la instrucció `new` directament perquè aquesta necessita conèixer en temps de compilació la classe de l'objecte que posteriorment haurà de crear.

Què fem, doncs?

> Podríem canviar la crida a `new` per una crida a una funció polimòrfica que generés dinàmicament una rèplica de l'objecte amb el qual es cridés.
>
> Com es tracta d'una funció polimòrfica, el tipus de l'objecte que s'ha de replicar es decidirà en temps d'execució.
>
> A aquesta funció l'anomenarem *clonació*.

Ens caldrà definir una funció *clonació* per cadascuna de les classes de les quals vulguem crear instàncies dinàmicament. Per acompanyar la funció *clonació*, en definirem una altra que s'encarregarà de fer una còpia d'un objecte: l'anomenarem *còpia*.

- `virtual T* clonació()`

  `x.clonacio()` crearà un *nou* objecte del mateix tipus que x, copiarà el valor de x a aquest objecte nou i retornarà un apuntador a ell.

  En definitiva, retornarà un apuntador a un objecte *nou* que contindrà la mateixa informació que x.

- `void copia(T&)`

  `x.copia(y)` col.locarà a l'objecte (ja existent) y una còpia de l'objecte x.

Veiem com es pot realitzar aquesta idea a les nostres classes familiars:

clonació és una funció polimòrfica (`virtual`) que es defineix a la classe *objecte_casa* i es redefineix a les classes *roba i aliment*. Per tant, el codi al que s'associarà cada crida a *clonació* dependrà del tipus real (en temps d'execució) de l'objecte sobre el qual es fa la crida.

Veiem-ho:

```
class objecte_casa {

 char lloc[MAXC];
 char marca[MAXC];
 char nom[MAXC];
 int preu;

public:

objecte_casa(){}

objecte_casa(char* plloc, char* pmarca, char* pnom, int ppreu);


virtual objecte_casa* clonacio()
{
   objecte_casa* aux;

   aux=new objecte_casa(lloc,marca,nom,preu);
   return aux;
}

void copia(objecte_casa& a)
{
```

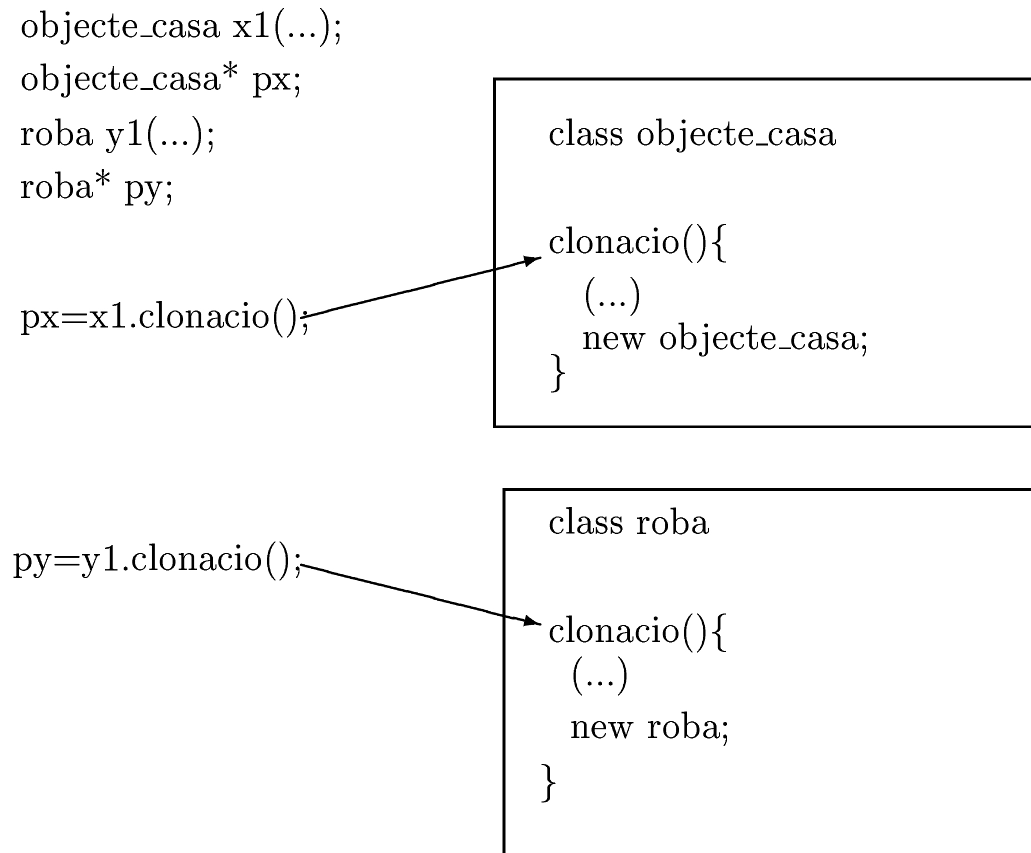


Figura 5.4: Funcionament de la funció *clonació*.

```
  a.preu=preu;
  strcpy(a.nom,nom);
  strcpy(a.marca,marca);
  strcpy(a.lloc,lloc);
}

//Altres operacions...

};
```

Dins la funció `objecte_casa::clonacio` coneixem exactament el tipus de l'objecte que caldrà crear, per tant, la crida a `new` és del tot correcta (veure figura 5.4).

Notem que per a que aquesta funció pugui ser polimòrfica, ha de ser virtual.

Veiem ara com definirem ambdues accions (clonació i còpia) en una subclasse de *objecte_casa*. En aquest codi veurem també la utilitat de l'acció *còpia*.

```
class roba :public objecte_casa {

 int talla;
 char material[MAXC];
```

```
public:

roba(){}

roba (char* plloc, char* pmarca, char* pnom, int ppreu, int ptalla,
      char* pmaterial);


virtual objecte_casa* clonacio()
{
  roba* aux;

  aux=new roba;            //[1]    ==> Crida implicita al constructor

  objecte_casa::copia(*aux);
  aux->talla=talla;
  strcpy(aux->material,material);

  return aux;

}

void copia (roba& b)
{
  objecte_casa::copia(b);
  b.talla=talla;
  strcpy(b.material,material);
}

//Altres operacions.

};
```

Amb la incorporació de la funció `roba::clonacio` obtenim el resultat esperat: si es genera la crida `x.clonacio()` i s'associa en temps d'execució al codi de `roba::clonacio()` vol dir que x era de classe *roba* i que es vol generar dinàmicament una instància d'aquesta classe.

En el disseny d'aquestes classes hi ha alguns detalls subtils que no hem de passar per alt:

- La classe *roba* necessita un constructor que no prengui paràmetres perquè a la crida a l'operador `new` ([1]) caldria passar-li els paràmetres del constructor, i no tots són accessibles (els privats d'*objecte_casa* no ho són (veure 4.6). Aquest és el motiu que fonamenta l'existència de l'operació `copia`.
- El tipus de retorn de la funció `roba::clonacio()` hauria de ser `roba*` (`roba* clonacio()`), però això planteja el problema de que `roba* roba::clonacio()` no redefineix `objecte_casa* objecte_casa::clonacio()`, com és el nostre desig[8].

  La definició de `objecte_casa* roba::clonacio()` no planteja cap problema perquè *roba* és un subtipus de *objecte_casa*.

[8] En realitat C++ entendria que una funció redefineix l'altra però que els tipus de retorn no concorden i mostraria un missatge d'error per aquesta circumstància.

5.6.3 La destrucció

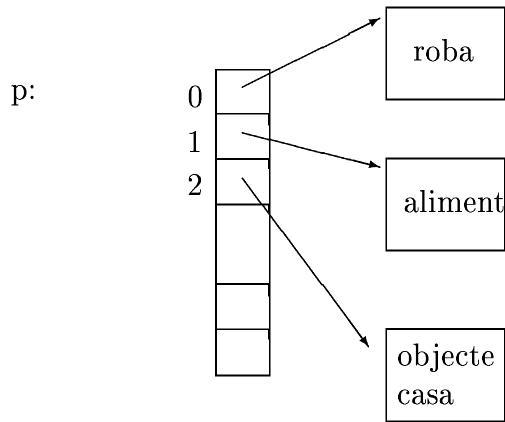
El disseny de la classe *pila d'objectes de la casa* quedaria incomplet si no la dotéssim d'una operació destructora. Veurem que això haurà d'anar acompanyat de la definició d'operacions destructores virtuals per a *objecte_casa* i les seves classes derivades.

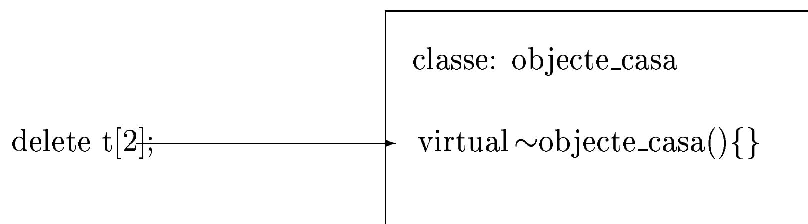
Si pensem en una operació destructora d'un objecte de tipus *pila d'objectes de la casa*, el més senzill és proposar una acció de l'estil:

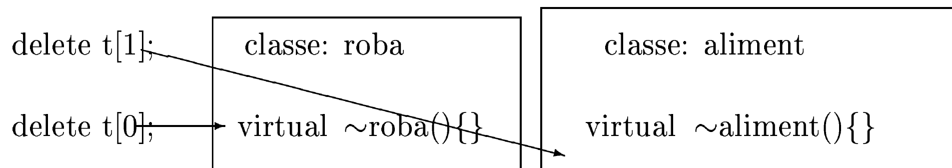
```
class pila_oc{

objecte_casa* t[MAX];
int top;

public:

//Resta d'operacions...

~pila_oc(){
   int i;

   for (i=0;i<top;i++) {
     delete t[i];
   }
 }
};
```

La instrucció `delete t[i];` cridarà *estàticament* el destructor (definit per defecte) de la classe *objecte de la casa* independentment del tipus real de cada element de la pila. Això donarà lloc a una destrucció incompleta dels elements de tipus *roba, aliment o medicament.* La solució serà definir destructors virtuals per les classes *objecte_casa*, *roba* i *aliment* (que tindran codis buits) i per la classe *medicament*, amb el codi que presentem més avall.

```
class objecte_casa{

//representacio

public:

virtual ~objecte_casa(){}

//Mes operacions

};

class aliment :public objecte_casa{

//representacio

public:
```







Si *objecte_casa* té definits destructors virtuals, `delete t[0]` alliberarà exactament el nombre de bytes que ocupa la representació d'una peça de roba, mentre que `delete t[1]` i `delete t[2]` faran el mateix respectivament amb els aliments i els objectes de la casa.

Figura 5.5: Destrucció d'una pila d'objectes de la casa

```
virtual ~aliment(){}

//Mes operacions

};

class roba :public objecte_casa{

//representacio

public:

virtual ~roba(){}

//Mes operacions

};


class medicament :public objecte_casa{

//representacio

public:

virtual ~medicament()
{
 delete [] malaltia;
 delete [] metge;
}

//Mes operacions

};
```

Ara ja estem en condicions de redissenyar la classe `pila_oc`:

5.6.4 La pila d'objectes de la casa

```
class pila_oc{

objecte_casa* t[MAX];
int top;

public:

pila_oc(){top=0;}

~pila_oc()
 {
   int i;
```

```
   for (i=0;i<top;i++) {
     delete t[i];
   }
}

void empilar(objecte_casa& x)
{

  t[top]=x.clonacio();
  top++;
}

void desempilar()
{
  top--;
  delete t[top];
}


objecte_casa* cim()
{
  objecte_casa* roc;

  roc=t[top-1]->clonacio();
  return roc;
}


char p_buida()
{
 return top==0;

}

};
```

L'acció `empilar` ara ja és correcta perquè efectivament empilarà una còpia del paràmetre x tenint en compte el tipus que té aquest paràmetre *en temps d'execució*. Si aquest tipus és *objecte_casa*, es farà la crida `objecte_casa::clonacio()`, es crearà un *nou objecte_casa* i s'empilarà. Si el tipus fos *roba*, es farà la crida `roba::clonacio()` i s'empilarà un nou objecte de tipus *roba*. El mateix passarà amb l'operació `cim`.

També ens podem plantejar enriquir la classe *pila_oc* amb l'operació `clonacio()` que tindria l'objectiu de generar una còpia nova de la pila sobre la qual es cridés. Després del treball que hem fet fins ara no és difícil pensar en una solució:

```
pila_oc* pila_oc::clonacio()
{
  int i;
  pila_oc* pc;
```

```
  pc=new pila_oc;

  pc->top=top;
  for(i=0;i<top;i++){
    pc->t[i]=t[i]->clonacio();
  }
  return pc;
 }
```

5.6.5 El client

Un possible programa client d'aquestes classes podria ser el següent:

```
main()
{

objecte_casa o1(...);
roba r1(...);

pila_oc p1;
pila_oc* p2;

p1.empilar(o1);
p1.empilar(r1);


p2=p1.clonacio();

cout << "pila p2:\n";

llistar(*p2);

}
```

Capítol 6

Les classes genèriques. Els `template`

6.1 Concepte

Si considerem les classes *pila de naturals*, *pila de caràcters*, *pila de reals, pila de complexos,...* haurem de convenir que totes elles han d'exhibir un comportament essencialment equivalent. Dit d'una altra manera, el comportament general d'una pila (empilar un element, desempliar un element, obtenir l'element del capdamunt d'una pila,...) no depén del tipus d'elements de què estigui formada la pila. Aquesta idea ens porta a pensar que segurament seria interessant considerar una classe *genèrica* pila, on es descrigués el comportament general dels objectes pila de forma independent del tipus particular dels seus elements. Aquest tipus particular seria un paràmetre de la classe genèrica pila.

La mateixa idea la podrem utilitzar per qualsevol altra classe que sigui una col.lecció d'objectes d'algun tipus determinat (llistes, taules, arbres, grafs...).

> Direm que una classe A és genèrica si la seva definició depén d'una o més classes (tipus) que actuen com a paràmetres de A.
>
> Definir una relació d'instanciació sobre una classe genèrica A (o simplement instanciar aquesta classe) vol dir donar un valor concret a la classe que actua com a paràmetre de A.
>
> L'eina que ofereix el llenguatge C++ per a definir relacions d'instanciació s'anomena `template`.
>
> Una *classe-template* és una classe que té algun tipus com a paràmetre i que actua com a *patró* o *plantilla* amb la qual instanciar classes concretes.

Amb el que hem explicat fins ara ja podem deduir que la definició d'una *classe-template* presenta diferències significatives respecte la definició d'una classe habitual. La diferència fonamental rau en el fet que una *classe-template* no és ben bé una classe sinó una *plantilla* d'una classe, un *patró* que contindrà les eines i les instruccions necessàries per a construir una classe amb tots els seus atributs i operacions, però no és pròpiament la classe. En particular, no podem definir directament objectes d'aquella classe-template. Primer l'hem d'instanciar.

Tot seguit posem una col.lecció d'exemples:

6.2 Exemple 1

```
template<class T>
class pila{
```

```
  T vec[MAX];
  int damunt;

public:

  pila() { damunt=0;}
  void empilar(T e) {vec[damunt]=e; damunt++;}
  void desempilar() {damunt--;}
  T cim() {return vec[damunt-1];}
 };


main()
{
  pila<int> p_enters(100);      //p_enters es una pila amb capacitat
//per a 100 enters.

  pila<objecte_casa> p_obj(40); //p_obj 'es una pila amb capacitat
//per a 40 objectes de la casa.
  int j=9;
  objecte_casa ob1("sala","pilips","televisor", 45000);


  p_enters.empilar(j);
  p_obj.empilar(ob1);

}
```

El codi d'aquest exemple que va des de l'inici fins la funció `main` és la definició del `template` (patró o plantilla) de la classe *pila*[1]. La primera declaració (`template<class T>`) indica que el que ve tot seguit no és pròpiament una classe sinó la plantilla d'una classe la qual dependrà d'un cert tipus *genèric* T. Efectivament, en la definició de *pila* ens trobem amb atributs (`T vec[MAX]`) o bé operacions (`void empilar(T e)...`) que utilitzen el tipus T que no ha estat definit enlloc sinó que és un paràmetre de la classe genèrica.

Mitjançant la instanciació d'aquest paràmetre T per una classe concreta (`int` o `objecte_casa`) creem una autèntica classe *pila* (no una plantilla de pila). Així,

`pila<int>` és una pila d'enters i

`pila<objecte_casa>` és una pila d'objectes de la casa.

La definició de la funció `main`: `pila<int> p_enters(100);`

genera, utilitzant la plantilla, la següent classe:

```
class pila{

  int vec[MAX];
  int damunt;
```

[1] Aquest exemple com tampoc la resta dels que presentem no contemplen possibles errors d'empilar en una pila plena o desempilar o consultar el cim d'una buida.

```
public:

  pila() { damunt=0;}
  void empilar(int e) {vec[damunt]=e; damunt++;}
  void desempilar() {damunt--;}
  int cim() {return vec[damunt-1];}
 };
```

I, seguidament, l'objecte `p_enters` com a instància d'aquesta classe amb tamany 100.

> No és fins el moment en què instanciem un *patró* (*template*) de classe amb un cert tipus concret que aquella classe adquireix existència pròpia. Fins aquell moment només tenim un *manual d'instruccions per a construir una classe.*

6.3 Exemple 2

A l'exemple anterior hem definit els codis de les operacions dins de la pròpia classe. Ja sabem que és habitual (sobretot quan aquests codis són llargs i complexos) definir-los fora. La forma de fer això quan estem treballant amb `templates` és la següent:

```
template<class T>
class pila{

  T vec[MAX];
  int damunt;

public:

  pila();
  void empilar(T e);
  void desempilar();
  T cim();
 };

template<class T> pila<T>::pila(int n)
{
  damunt=0;
}

template<class T> void pila<T>::empilar(T e)
{
  vec[damunt]=e;
  damunt++;
}

template<class T> T pila<T>::cim()
{
  return vec[damunt];
}
```

6.4 Exemple 3

Suposem que volem enriquir la classe genèrica *pila* que ens ha servit com a exemple fins ara amb les operacions `bool cerca_element(T x)` (que cercarà l'element x dins la pila) i `void llista_elements()` (que llistarà tots els elements que contingui la pila). Aquestes dues operacions addicionals, convertiran la classe *pila* en una classe *pila recorrible.* L'anomenarem així perquè mantenim l'estratègia *LIFO* que caracteritza les piles (no modifiquem per a res *empilar* i *desempilar*) però ara, a través de les dues operacions afegides, tenim accés a tots els elements de la pila i no només al primer com fins ara.

La incorporació d'aquestes dues operacions posa en evidència un problema important. Proposem una possible implementació per a fer-lo explícit:

```
template<class T> bool cerca_element (T x)
{
    int i=0;
    bool trobat=FALS;

    while (!trobat && i<damunt){
      trobat=(vec[i] ``es igual a'' x);  [1]
      i++;
    }
    return trobat;
}


template<class T> void llista_elements ()
{
    int i=0;

    while (i<damunt){
      ``llistar el valor de vec[i]'';   [2]
      i++;
    }
}
```

En la implementació d'aquestes dues operacions veiem que necessitem una funció per a comparar dos elements de classe T i una altra per a escriure un element de classe T. Però com T és, en realitat, un paràmetre que més endavant es podrà instanciar amb *qualsevol* classe, les funcions que triem hauran d'estar definides (amb el mateix nom) en qualsevol classe que pugui instanciar el paràmetre T. Aquesta reflexió ens suggereix la següent solució:

> Utilitzar els operadors usuals d'igualtat (==) i de sortida (<<) de C++, sobrecarregant-los per cadascuna de les classes amb què es pugui instanciar T per a que tinguin el significat específic requerit per aquella classe

Les línies [1] i [2] quedarien ara de la següent manera:

```
[1]:      trobat=(vec[i] ==x);


[2]:       cout << vec[i];
```

Com a conseqüència, la classe genèrica *pila_recorrible* només es podrà instanciar amb aquelles classes que tinguin definits (predefinits o sobrecarregats) els operadors d'igualtat (==), de sortida (<<) i d'assignació (=)[2 3].

A l'apartat 8 es presenta la sobrecàrrega d'operadors i de funcions.

6.5 Exemple 4

Els grafs són estructures de dades que es defineixen utilitzant dos conjunts: un conjunt de vèrtexs i un altre d'arestes (que representen parells de vèrtexs relacionats). Els grafs s'utilitzen per a modelar diferents situacions. En cada situació, el significat dels vèrtexs i de les arestes pot ser diferent. Si es vol modelar una xarxa de carreteres, per exemple, els vèrtexs correspondrien a ciutats (que podrien tenir una sèrie d'informació associada com el nom, nombre d'habitants...) i les arestes a carreteres (amb informacions tals com el nom de la carretera, la distància entre les dues ciutats...).

Podem pensar en una classe genèrica *graf* que dependria ara, no d'un, sinó de dos paràmetres *tipus*: els *vèrtexs* i les *arestes*. Vegem com ho implementaríem en C++:

```
template<class vertex, class aresta>
class graf{

//Representaci'o triada


public:

graf(llista<vertex>);

//Construeix un graf com una llista de vertexs i cap aresta.

void afegir_aresta(vertex, vertex, aresta);

//Afegeix una aresta entre dos vertexs.

// M'es operacions

};

class ciutat{


};
```

[2] L'operació `empilar` l'utilitza.

[3] En realitat cada classe en C++ defineix per defecte un constructor sense paràmetres, un constructor copiador, l'operador d'assignació (=) i el d'igualtat (==). Però és possible que aquesta definició per defecte no s'ajusti als nostres requeriments.

```
class carretera{

};

//(...)

llista<ciutat> lciu;

//(...)

omplir_llista(lciu);

//(...)

graf<ciutat,carretera> xarxa_carr(lciu);
```

- La sobrecàrrega d'operadors es presenta a l'apartat 8.

Capítol 7

Les funcions `friend`

Imaginem que tenim dues classes, la classe *conjunt d'enters* i la classe *llista d'enters* amb les següents operacions:

```
class llista{

//...

public:

llista();
void inserir_elem(int x);
void anar_al_primer_elem();
void avancar_un_elem();
int obtenir_elem_actual();
};

class conjunt{

//...

public:

conjunt();
void inserir(int x);
int pertany_elem_a_cjt(int x);
int conjunt_buit();
};
```

Considerem ara una nova funció, amb l'*especificació* següent:

$$mateixos_elements(conjunt, llista) \longrightarrow boolea$$

Volem que aquesta nova funció retorni *cert* si el conjunt i la llista que se li passen per paràmetre contenen els mateixos elements i $fals$ en cas contrari.

En aquest punt ens sorgeix un petit problema: A quina de les dues classes posem aquesta nova funció? Com la programem?

Fem alguna consideració al respecte:

- Aquesta funció no pertany de forma natural ni a *llista* ni a *conjunt* ja que no està dissenyada per a ser aplicada de forma predominant sobre un objecte de classe *llista* ni tampoc sobre un de classe *conjunt* sinó més aviat sobre un objecte de cada classe i simultàniament[1].
- Com a conseqüència de l'anterior consideració podríem pensar que *mateixos_elements* hauria de ser una funció *client* d'ambdues classes. D'aquesta manera, utilitzant les operacions de la interfície de totes dues classes podríem anar recorrent els elements de la llista i els del conjunt tot i cercant alguna diferència. Si no hi hagués cap diferència, els dos objectes contindrien els mateixos elements i la funció retornaria *cert*. En cas contari, hauríem trobat una diferència i la funció retornaria *fals*.

  Aquesta aproximació presenta, malauradament, el problema que la classe conjunt no disposa de cap operació per a recórrer els seus elements (recordem que, en realitat, un conjunt no és una estructura de dades seqüencial i, per tant, no hi està obligat).

En conclusió, ens trobem davant d'una funció que no és aconsellable incorporar com a membre a cap de les dues classes presentades però que tampoc pot ser client d'elles (almenys de forma senzilla).

C++ proposa una eina per a sortir d'aquest *cul-de-sac*: considerar una nova categoria de funcions que, sense ser membres d'una determinada classe, tenen accés a la part privada d'aquella classe.

> Una funció *friend* d'una sèrie de classes $A_1,\ A_2, ..., A_n$ és una funció tal que:
>
> - No és membre de cap d'aquelles classes.
> - Pot accedir a la part privada de totes elles.

mateixos_elements és un candidat idoni per a ser funció *friend* de les classes *llista* i *conjunt*: no serà membre de cap de les dues classes (i així superem la primera de les consideracions) però podrà accedir a les parts privades (a la implementació) de totes dues (per la qual cosa no haurà d'actuar de client i podrà accedir fàcilment a tots els elements del conjunt).

La forma d'incorporar funcions *friend* en C++ és la següent:

```
class llista{

//...

public:

llista();
void inserir_elem(int x);
void anar_al_primer_elem();
void avancar_un_elem();
int obtenir_elem_actual();
friend int iguals_elements(conjunt& c, llista& l);
};

class conjunt{
```

[1] Amb aquesta apreciació no estem dient que no sigui possible incorporar aquesta funció a alguna de les dues classes sinó més aviat que no és massa natural, ni adequat, ni convenient fer-ho.

```
//...

public:

conjunt();
void inserir(int x);
int pertany_elem_a_cjt(int x);
int conjunt_buit();
friend int iguals_elements(conjunt& c, llista& l);
};


int iguals_elements(conjunt& c, llista& l)
{...}
```

Hem d'insistir en que, malgrat a dins de les dues classes hi aparegui la capcelera de *iguals_elements*, aquesta funció *no és membre* de cap de les dues classes.

Hi ha una prevenció que cal fer respecte l'ús (i més encara, l'ús indiscriminat) de les funcions *friend*: dos dels puntals més importants sobre els quals descansa la POO són l'abstracció i l'encapsulament: una classe ha de quedar definida vers els seus clients en termes dels serveis que ofereix i fora de la classe ningú no pot accedir a les seves intimitats (la seva implementació).

Les funcions *friend* trenquen aquesta aproximació i, per tant, s'haurà de dosificar molt el seu ús limitant-lo a situacions en les que realment resultin molt recomanables.

En aquest text recomanem utilitzar les funcions *friend* fonamentalment en dos casos:

1. Si no resulta natural incorporar una funció com a membre d'una classe però, per altra banda, o no és possible convertir-la en client de la mateixa o bé resulta molt ineficient.

2. Per sobrecarregar determinats operadors (veure 8).

Capítol 8

La sobrecàrrega de funcions i operadors

8.1 Concepte

Existeixen operacions, com ara l'assignació, la comparació per igualtat, la lectura i l'escriptura d'objectes o, fins i tot, la suma, la resta o el producte que són comunes a moltes classes, tant a classes predefinides (`int`, `char*` ...) com a altres definides per l'usuari (*complex*, *objecte_casa*...). Per tant, serà freqüent que a l'hora de definir una nova classe ens trobem amb la necessitat de definir alguna de les operacions anteriors per aquesta classe.

Davant d'aquesta necessitat podem fer dues coses:

- Inventar una operació nova amb un nom nou(*assigna_complex*, *assigna_oc*...)
- Conservar el nom de l'operador tradicional i dotar-lo d'un nou significat. Per tant, tindríem el mateix nom d'operador = per a assignar complexos i objectes de la casa (i també enters, reals...).

Si optem per la segona forma d'actuació diem que estem sobrecarregant l'operador d'assignació.

> Sobrecarregar una funció consisteix en crear una funció que tingui el mateix nom d'una altra ja existent però paràmetres de diferent tipus i un comportament també, en general, diferent.
>
> En general, el compilador s'encarrega de decidir quin és el codi que cal associar a cada crida d'una funció sobrecarregada (lligam estàtic). Aquesta decisió depén dels tipus dels paràmetres de la crida.
>
> En el cas de les funcions polimòrfiques (virtuals) sobrecarregades, el compilador no pot decidir quin codi associar a cada crida. En aquest cas, el lligam es fa en temps d'execució i s'anomena *lligam dinàmic o tardà* (veure 5).

La sobrecàrrega de funcions és la manera millor i més elegant d'enfrontar-se a la situació d'haver de redefinir comportaments de funcions. De fet, aquest concepte no és pas nou: utilitzem sobrecàrrega en aritmètica elemental quan fem servir els mateixos símbols d'operadors (+, −...) per a operar amb diferents conjunts (enters, reals, complexos...) i utilitzant diferents algorismes.

Els criteris que recomanen la sobrecàrrega són almenys tres:

- Simplicitat. Evitem la generació d'una quantitat immensa de funcions que, desd'un punt de vista abstracte, fan el mateix.

- Universalitat. Els mateixos operadors estan definits per a totes les classes (especialment important per les classes genèriques. Veure 6).
- Conservació de la notació. La sobrecàrrega ens permetrà seguir treballant amb la notació infixa que ens és més familiar (podrem seguir escrivint $c = a + b$).

8.2 Sobrecàrrega de funcions

Ja n'hem vist exemples a 4 i a 5.

Aquells exemples, en realitat, feien referència a un tipus més estricte de sobrecàrrega que s'anomena *redefinició*.

Dues funcions sobrecarregades tenen el mateix nom però, en general, paràmetres diferents.

Una funció que redefineix a una altra té el mateix nom i el mateix tipus (o subtipus) dels paràmetres.

8.3 Sobrecàrrega d'operadors

Com ja hem discutit, en general, les funcions que se sobrecarreguen amb més freqüència són els *operadors* predefinits $(+, -, *, =, ==, << ...)$. La seva sobrecàrrega ens permet, d'una banda, utilitzar operadors universals per a referir-nos a operacions conceptualment semblants (encara que es desenvolupin sobre tipus diferents) i, d'altra, mantenir la notació infixa que acostumem a utilitzar amb ells.

Existeixen dues possibilitats de sobrecarregar operadors: utilitzant membres de la pròpia classe o bé utilitzant funcions *friend*.

8.3.1 Sobrecàrrega d'operadors amb membres de la pròpia classe

Considem la classe *complex*. Aquesta classe necessitarà redefinir els operadors de $+, -, *, /, =, ==, << ....$

Entre tots aquests operadors, considerem el d'assignació: =.

Es pot entendre que aquest operador *s'aplica sobre l'objecte que és modificat i pren com argument l'objecte, el valor del qual s'assigna al primer.* Des d'aquest punt de vista queda perfectament clar que l'operador d'assignació segueix *fil per randa* la filosofia de les operacions de les classes en la P.O.O. (recordem que, segons aquesta filosofia, les operacions s'apliquen sobre els objectes tot i modificant-los o consultant-los). Per tant, proposarem que l'operador d'assignació (=) se sobrecarregui com a operació de la classe *complex*. Ho farem de la següent manera:

```
class complex{

  double real;
  double imagin;

public:

 complex(double r, double i)
   :real(r), imagin(i){}

 void operator=(complex c){ real=c.real; imagin=c.imagin;}
```

```
//Altres operacions.
};
```

Aquest operador sobrecarregat es podrà utilitzar a partir d'ara seguint la mateixa notació amb la qual fem assignacions sobre tipus predefinits:

```
main()
{

 complex c1(2,3), c2(0,0);

 c2=c1; //crida a l'operador d'assignacio '='
//sobrecarregat.

}
```

Un operador sobrecarregat com a membre d'una classe es pot definir com a virtual.

8.4 Sobrecàrrega d'operadors amb funcions *friend*

Considerem altre cop la classe *complex.* Centrem-nos ara amb un altre operador, en aquest cas, l'operador +. La forma usual de treballar amb ell divergeix de la filosofia de la POO per a les operacions. Mentre que aquesta filosofia es basa en aplicar operacions a un objecte per a modificar-lo o consultar-lo, l'operador + consulta dos objectes i en retorna un de nou, que té com a valor la suma dels dos anteriors. Per tant, l'operador + no és natural veure'l com a modificador de l'objecte sobre el qual s'aplica i el mateix es pot dir dels operadors (−, ∗, ==, <<...). En aquests casos, el millor és sobrecarregar utilitzant funcions *friend*:

```
class complex{

  double real;
  double imagin;

public:

 complex(double r, double i)
   :real(r), imagin(i){}

 void operator=(complex c){ real=c.real; imagin=c.imagin;}

 friend complex operator+(complex, complex);
 friend complex operator-(complex, complex);
 friend complex operator*(complex, complex);
 friend int operator==(complex, complex);
```

```
//Mes operacions...

};


complex operator+(complex c1, complex c2)
{

  complex aux;

  aux.real=c1.real+c2.real;
  aux.imagin=c1.imagin+c2.imagin;

  return aux;

}

//...


main()
{

  complex x1(2,3), x2(5,7), x3(0,0);


  x3=x1+x2;
}
```

Els operadors $+$, $-$, $*$ i / podrien ser sobrecarregats com a funcions membres de la classe complex encara que el resultat no fóra tan natural:

```
class complex{

//...

public:
//...
complex operator+(complex c){
  complex aux;

  aux.real=real+c.real;
  aux.imagin=imagin+c.imagin;
  return aux;
}
//...

};
```

```
main()
{

 complex c1(2,3),c2(4,5),c3(0,0);


 c3=c1+c2; //equivalent a c3=c1.operator+(c2);

}
```

Notem finalment que només passem un argument perquè l'altre és l'objecte sobre el qual es crida l'operació.

8.5 La sobrecàrrega de l'operador <<

A l'apartat anterior hem incidit en dos aspectes fonamentals respecte a la sobrecàrrega d'operadors:

- Una manera molt natural de sobrecarregar determinats operadors com ara == o << consisteix en fer-ho a través de funcions *friend*.
- Els operadors que sobrecarreguem com a membres de la classe els podem declarar com a virtuals (i, per tant, convertir-los en operadors polimòrfics).

D'aquestes dues consideracions es deprén ben ràpidament una pregunta important: *es pot declarar virtual un operador sobrecarregat utilitzant una funció friend?*

La resposta és *no* perquè les úniques funcions que poden ser virtuals són les membres d'una classe i una funció friend no és membre de cap classe (veure 5.2 i 7).

Una conseqüència pràctica d'aquest fet és que l'operador << no pot ser virtual (recordem que aquest operador es definia mitjançant una funció *friend*). Això sembla una mala notícia per totes aquelles classes que requereixen un operador de sortida virtual (com ara *objecte_casa*) ja que, per elles, no podrem sobrecarregar l'operador *universal* << tot i definint-lo com a `virtual`.

A hores d'ara, els més pessimistes poden pensar que ens veiem obligats a mantenir l'operació *llista_caract()* com a única operació de sortida de la classe *objecte_casa* i derivades. Existeix, però, una senzilla solució que podem adoptar i que resol satisfactòriament el problema. Es tracta de definir una funció *friend* que sobrecarregui l'operador << amb el paràmetre de tipus *objecte_casa* passat per referència que contingui una crida a la funció polimòrfica *llista_caract()*.

```
ostream& operator<<(ostream& c, objecte_casa& oc)
{
  oc.llista_caract();
  return c;
}
```

Aquesta funció cridarà a `objecte_casa::llista_caract()` o a `roba::llista_caract()` depenent del tipus dinàmic (en temps d'execució) de l'objecte `oc`.

Evidentment caldrà definir aquesta funció com a *friend* d'*objecte_casa*.

```
class objecte_casa{

 //Representacio...

public:

friend ostream& operator<<(ostream& , objecte_casa&);

//Altres operacions.

};
```

No és necessari definir-la com a *friend* de les subclasses d'*objecte_casa* perquè la mateixa funció es cridarà per a qualsevol subclasse d'objecte de la casa (perquè tant *aliment* com *roba* só subtipus d'*objecte_casa*).

8.6 Detalls addicionals

La sobrecàrrega de funcions i operadors té una importància molt gran quan es treballa amb templates. Efectivament, en la definició d'una classe *template* gairebé sempre cal fer alguna hipòtesi sobre el tipus genèric que serveix com a paràmetre de la classe. Les hipòtesis més usuals consisteixen en suposar que el tipus genèric té definides algunes operacions tals com (l'assignació, la comparació per igualtat, per més gran o més petit...). En aquests casos és útil anomenar aquestes operacions de la forma usual (=, ==...) i definir-les als tipus que instanciaran el template també de la mateixa forma. D'aquesta manera obtindrem *operadors universals*, vàlids en la gran majoria de les classes (tan predefinides com definides per l'usuari).

Els operadors d'assignació (=) i d'igualtat (==) es defineixen per defecte. El significat per defecte és el d'assignar i comparar per igualtat respectivament els membres de la classe definits estàticament. Aquest comportament pot no ser el desitjat, (sobretot si es treballa en memòria dinàmica. Veure 2.10) i, en general, caldrà sobrecarregar-los.

Els operadors sobrecarregats admenten arguments per referència.

Per exemple:

```
friend complex operator+(complex&, complex&);
```

Capítol 9

E/S en C++

9.1 Introducció a l'E/S en C++

9.1.1 Fluxos

Els mecanismes d'entrada/sortida (E/S) d'un llenguatge de programació descriuen la manera com un programa escrit en aquell llenguatge pot llegir (escriure) dades de (a) algun dispositiu extern. Hi ha diversos tipus de dispositius externs (pensem, per exemple, en el monitor, el teclat o els fitxers), per tant, un llenguatge de programació, com ara el C++, pot optar per definir una manera diferent de gestionar l'E/S per cada tipus particular de dispositiu o bé, al contrari, definir una manera *uniforme* de manejar l'E/S que resulti escaient per qualsevol dispositiu. Ja es veu que aquesta segona opció és molt més simple i elegant perquè permet treballar d'una manera uniforme amb independència del dispositiu concret amb el que fem E/S.

Per tal d'aconseguir aquesta uniformitat C++ defineix un *dispositiu abstracte* que anomena *flux* (*stream*). Intuïtivament un flux és una seqüència de *bytes* de longitud indefinida a (de) la qual es poden escriure (llegir) bytes. Un programa escrit en C++ podrà definir objectes de tipus *flux* (això és, objectes definits d'alguna de les classes derivades de *ios*, que és la classe arrel de la jerarquia que gestiona els fluxos)[1]. Aquests objectes s'associaran a dispositius concrets (com ara l'entrada estàndar o el fitxer *dades.dat*). Cada cop que el programa vulgui fer lectures o escriptures sobre aquell dispositiu concret, les farà sobre el flux al qual està associat. Un cop determinat un conjunt d'operacions per fer E/S sobre fluxos, haurem definit una manera estàndar i uniforme per tal gestionar E/S amb independència del dispositiu concret.

> Un flux és un objecte del programa d'una classe derivada de *ios*[2] que es pot lligar a un dispositiu d'E/S concret i sobre el qual es fan les operacions d'E/S adreçades a aquell dispositiu. Aquestes operacions són uniformes per treballar amb qualsevol dispositiu i vénen definides per les subclasses de la classe *ios*.

Com un flux és un objecte d'una classe determinada, el programa pot crear tants fluxos com necessiti per tal d'accedir als diferents dispositius amb què treballa (particularment fitxers). Ara bé, hi ha quatre fluxos creats per defecte i que el programa podrà utilitzar sense haver de definir. Són *cin, cout, cerr* i *clog*.

Els fluxos *cin*, *cout*, *cerr* i *clog* estan creats per defecte i tenen el sentit següent:

- **cin** fa referència a l'entrada estàndar (la qual, usualment està associada al teclat).

[1] Vegeu 9.1.3
[2] Vegeu 9.1.3

- **cout** fa referència a la sortida estàndar (usualment associada al monitor).
- **cerr** fa referència al flux al qual s'adreçaran els missatges d'error de forma estàndar. Usualment, com *cout*, està associat al monitor.
- **clog** fa referència al flux al qual s'adreçaran els missatges d'error a través de buffer.

Completarem aquesta secció introductòria presentant quines són les operacions fonamentals per tal de fer E/S sobre fluxos i quina és la jerarquia de classes definides per C++ per treballar amb fluxos.

9.1.2 Operacions bàsiques sobre fluxos

Les operacions essencials sobre fluxos són la d'*inserció* de caràcters a un flux (operador <<) i la d'*extracció* de caràcters d'un flux (operador >>). Aquests operadors estan definits per tots els tipus bàsics de C++ i podran ser sobrecarregats per totes les classes definides pel programador i, per tant, proveiran una interfície estàndar d'E/S en C++ aplicable a qualsevol classe.

L'operador d'inserció (<<)

Aquest operador també s'anomena *operador de sortida* i permet fer una operació de sortida sobre el dispositiu al que està associat el flux sobre el qual s'aplica l'operador[3].

L'operador << està definit per tots els tipus bàsics i es pot sobrecarregar per totes les classes definides per l'usuari. A ??? es pot veure la manera de sobrecarregar aquest operador.

Aquest operador se sol aplicar sobre els fluxos predefinits *cout* (sortida estàndar) i *cerr* (sortida pels errors) i també sobre fluxos associats a fitxers (vegeu 9.6).

Exemple 9.1.1
```
#include <iostream.h>

void main()
{
 cout << "hola";
}
```
Aquest programa envia la cadena "hola" a la sortida estàndar (recordem que cout és un flux associat de manera predefinida a la sortida estàndar). La necessitat de la inclusió del fitxer iostream.h s'explica a 9.1.3.

Exemple 9.1.2
```
#include <iostream.h>

void main()
{
 int i=90;
 float x=8.5;

 cout << "valor de i=" << i <<"\n"
      << "valor de x="<< x <<endl;
}
```

En aquest exemple es podem fer tres observacions diferents:

[3]Notem que aqeust operador suggereix la forma d'una fletxa que indica el sentit en què es mouen les dades (en aquest cas cap al flux).

1. L'operador << és associatiu a l'esquerra. Això vol dir que la instrucció de sortida

   ```
   cout << "valor de i="<< i;
   ```

   és equivalent a:

   ```
   (cout << "valor de i=") << i;
   ```

   Per a que aquest comportament associatiu sigui possible és necessari que l'operador << retorni com a resultat el flux sobre el qual s'ha aplicat (en aquest cas *cout*). D'aquesta manera, la següent aplicació de l'operador << es farà sobre el flux que la primera apliació ha retornat. La definició d'aquest operador és:

   ```
   ostream& operator << (T);
   ```

   on T és algun dels tipus bàsics.

   I, segons aquesta definició, l'operador << retorna un flux (de classe *ostream*) tal com és requerit.

2. *endl* és un acrònim de *end line* que produeix un canvi de línia en la sortida (com '\n').

   endl serveix, a més, per una altra cosa: l'escriptura d'un element al dispositiu de sortida es gestiona via *buffer* per raons d'eficiència. Això vol dir que, quan s'ha ordenat l'escriptura de varis caràcters sobre un flux (mitjançant l'operador <<, per exemple) s'envien *d'un sol cop* tots aquells caràcters al dispositiu associat al flux. Per tant, hi pot haver un desfasament entre l'execució d'una instrucció amb l'operador << i l'escriptura real al dispositiu. *endl* força l'escriptura al dispositiu de tots els caràcters del *buffer* pendents de ser enviats. Una altra manera d'aconseguir això mateix és mitjançant l'operació *flush* com veurem a 9.3.

3. L'operador << està sobrecarregat. Notem que l'apliquem a un element de tipus *int* i a un altre de tipus *float*.

L'operador d'extracció (>>)

Aquest operador també s'anomena *operador d'entrada* i permet fer una operació de lectura del dispositiu al que està associat el flux sobre el qual s'aplica l'operador.[4].

Com l'operador d'inserció (<<), també el d'extracció està predefinit per tots els tipus bàsics i es pot sobrecarregar per totes les classes que el programador pugui definir. I també es pot comportar associativament.

Aquest operador se sol aplicar sobre el flux predefinit *cin* (entrada estàndar) i sobre fluxos associats a fitxers (vegeu 9.6).

Veiem un exemple d'utilització.

Exemple 9.1.3

```
#include <iostream.h>

void main()
{
 int i;
 float x;
 char s[10];

 cin>>i>>x>>s;
 cout<<i<<"\n"<<x<<"\n"<<s<<"\n";
```

[4]Com en el cas anterior, aquest operador suggereix la forma d'una fletxa que indica el sentit en què es mouen les dades (en aquest cas des del flux).

```
}
```

Aquest programa llegeix de l'entrada estàndar una cadena de caràcters que interpretarà com un valor enter (el qual col.loca a la variable i), una altra cadena de caràcters que interpretarà com un valor real (i el col.loca a x) i una darrera cadena de caràcters (que col.loca directament a s després d'afegir-hi la marca de final '\ 0'). Seguidament escriu tots tres elements per la sortida estàndar.

En concret, l'operador >> llegeix els següents caràcters *significatius* del flux (en aquest cas de *cin*) que encara no havien estat consumits.

Entenem per *caràcters significatius* els que no són separadors (' ', TAB, '\n', EOF,...). Per tant >> desprecia els caràcters separadors inicials, llegeix els significatius (i els interpreta segons el tipus de la variable que ha d'emmagatzemar la informació llegida) i acaba la lectura quan troba algun altre caràcter separador.

Així doncs, una possible seqüència de caràcters que l'usuari podria teclejar en l'execució del programa anterior seria la següent:

```
12 4.5 pepet
```

La qual cosa provocaria la sortida següent:

```
12
4.5
pepet
```

Com a conseqüència de l'algorisme de lectura que utilitza >>, aquest operador no serà massa apropiat per llegir (i assignar a una sola variable) una cadena de caràcters que contingui separadors. Fem un exemple:

Exemple 9.1.4

```
#include <iostream.h>

struct pallasso{

 char nom[60];
 int edat;

};

void main()
{

 pallasso p;

 cin>>p.nom>>p.edat;

 cout<<"\n dades llegides:\n";
 cout<<"nom="<<p.nom<<"\n"<<"edat="<<p.edat<<"\n";
}
```

Si teclegem:

```
jordi
23
```

tot anirà bé. Ara bé, si teclegem:

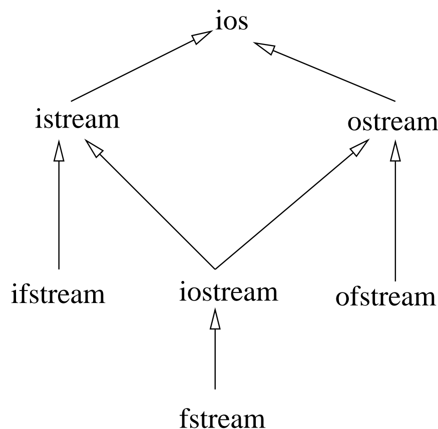


Figura 9.1: La jerarquia simplificada de classes d'E/S en C++

```
jordi ferrussol i solei
23
```

p.nom contindrà la cadena "jordi" mentre que p.edat contindrà la cadena "ferrussol", interpretada com un enter.

Degut a aquest comportament, sovint utilitzarem altres operacions per tal de llegir cadenes de caràcters (vegeu 9.2 i 9.3).

9.1.3 Jerarquia de classes

La figura 9.1 conté la jerarquia simplificada de les classes que gestionen els fluxos (i, per tant, les operacions E/S sobre dispositius) en C++.

- **La classe ios**

  És l'arrel de la jerarquia de classes d'E/S en C++. Bàsicament s'encarrega de:

  1. Controlar el format (això és, l'aparença visual) de l'entrada i la sortida. Els aspectes de format en l'E/S es discuteixen a 9.4.
  2. Guardar l'estat del flux en un moment determinat. L'estat d'un flux es discuteix a 9.5.
  3. Definir les modalitats d'accés a un fitxer (lectura, escriptura... Vegeu 9.6).

  La classe *ios* està definida al fitxer *streambuf.h.*

- **La classe istream**

  istream és una subclasse de *ios* que s'encarrega de gestionar els fluxos d'entrada. Per aquest motiu defineix mètodes (com ara *get, getline, read,* >>, ...) per a fer possible l'entrada d'informació des d'un flux. Per suposat hereta els elements de *ios* que permeten fer E/S amb formats (vegeu 9.4) i consultar l'estat d'un flux (vegeu 9.5). Aquesta classe és útil quan cal fer operacions d'entrada des d'un flux i es troba definida al fitxer *iostream.h.* Es presenta amb més detall a 9.2.

- **La classe ostream**

  ostream és una subclasse de *ios* que s'encarrega de gestionar els fluxos de sortida. Per aquest motiu defineix mètodes (com ara *put, write, flush,* <<, ...) per a fer possible la transferència (sortida) d'informació a un flux. Hereta els elements de *ios* que permeten fer E/S amb formats (vegeu 9.4) i consultar l'estat d'un flux (vegeu 9.5). Aquesta classe és útil quan cal fer operacions de sortida cap a un flux i es troba definida al fitxer *iostream.h.* Es presenta amb més detall a 9.3.

- **La classe iostream**

  Aquesta classe és subclasse, al mateix temps, de *istream* i de *ostream* (és tracta d'un exemple d'herència múltiple). Per aquest motiu, sobre un flux d'aquesta classe es poden utilitzar igualment les operacions d'entrada i les de sortida. Es troba definida al fitxer *iostream.h.*

- **La classe ifstream**

  La classe *ifstream* permet realitzar operacions de lectura de fitxers com si fossin fluxos d'entrada. Per a que això sigui possible aquesta classe defineix operacions per tal de lligar un flux a un fitxer (essencialment *open* i *close*) i hereta les operacions d'entrada definides a la classe *istream* 8de la que és subclasse). D'aquesta manera, primer de tot, associarem un fitxer a un flux del programa (*open*) i utilitzarem les operacions d'entrada habituals (>>, *get, getline...*) sobre aquest flux. Quan haurem acabat la lectura del fitxer el dissociarem del flux (*close). Notem que d'aquesta manera aconseguim uniformitat a l'hora de gestionar les entrades i les sortides sobre els diferents tipus de dispositius.*

  Aquesta classe es defineix al fitxer fstream.h i es presenta amb més detall a 9.6.

- **La classe ofstream**

  Aquesta classe permet realitzar operacions d'escriptura sobre un fitxer associant-lo a un flux de sortida. Com en el cas anterior, ofstream defineix operacions per tal de lligar un flux a un fitxer i hereta les operacions definides a la classe ostream (de la qual és subclasse) per gestionar la sortida.

  Aquesta classe es troba definida al fitxer fstream.h i es presenta amb més detall a 9.6.

- **La classe fstream**

  Seguint la mateixa filosofia de ifstream i ofstream permet fer operacions de lectura i d'escriptura sobre fitxers (que hauran estat prèviament associats a un flux).

  Aquesta classe es troba definida al fitxer fstream.h i es presenta amb més detall a 9.6.

9.2 Fluxos d'entrada (classe *istream*)

La classe *istream* és una subclasse de *ios* que sobrecarrega l'operador d'extracció >> per tots els tipus base i, a més a més defineix altres operacions relacionades amb la lectura d'informació d'un flux. Aquestes operacions són les següents:

- *get*

  Obté una cadena de caràcters del flux que s'està llegint (aquell sobre el qual s'aplica *get*). A diferència de >>, llegeix també caràcters separadors.

  En realitat no hi ha una única funció *get* sinó vàries sobrecarregades amb diferent propotipus:

  - `get(char& c);`

    Llegeix el següent caràcter del flux i el col.loca a la variable *c*.

    Exemple 9.2.1 `#include <iostream.h>`

    ```
    void main()
    {
     char c,d;


     cin.get(c);
     cin.get(d);
    ```

```
 cout<<"He llegit els caracters "<<c<<" i "<<d<<"\n";

}
```

– `get(char* cadena, int mida, int marcaFi='\n');`

Llegeix caràcters del flux d'entrada fins que es produeixi un dels tres fets següents:

* S'han llegit $mida - 1$ caràcters del flux o bé
* S'ha llegit el caràcter *marcaFi* (que, per defecte és '\ n', però pot ser qualsevol altre) o bé
* S'ha arribat a la fi del fitxer.

Exemple 9.2.2

```
void main()
{
 char c[20], d;

 cin.get(c,10,'t');
 cin.get(d);

  cout<<"He llegit els caracters "<<c<<" i "<<d<<"\n";
}
```

Si l'usuari entra:

`1234t5678`

la sortida serà

`He llegit els caracters 1234 i t`

Si l'usuari entra:

`12345678901234567890`

la sortida serà

`He llegit els caracters 123456789 i 0`

Notem que *get* col.loca '\0' al final de la cadena llegida per a formar d'aquesta manera una cadena de caràcters correcta en $C++$.

Notem també que la marca de final queda al flux (no s'incorpora a la cadena llegida).

Finalment, si es fa una crida a *get* sense el darrer argument (la marca de final) s'entén que la marca de final és '$\backslash n$'. Exemple:

`cin.get(s,20);`

- *getline*

  Funciona com *get* però afegeix la marca de final a la cadena llegida (i, per tant, l'elimina del flux).

- *ignore*

  `istream& ignore(int n=1, int separador=EOF);`

  Salta (i ignora) els propers caràcters del flux fins que passi algun dels dos fets següents:

  – S'han saltat n caràcters. El primer caràcter no llegit del flux és el número $n+1$. O bé

  – S'ha llegit el separador (que per defecte és EOF). El primer caràcter no llegit del flux és el que es troba immediatament després del separador.

  Si manca algun dels dos arguments (o tots dos) es pren per defecte $n = 1$ i $separador = EOF$.

- *putback*

  `istream& putback (char c)`

  col.loca a l'inici del flux el caràcter *c*, de forma que aquest serà el següent que es llegirà.

- *peek*

  `int peek()`

  Retorna el següent caràcter del flux sense extreure'l del flux (això vol dir que la següent operació de lectura sobre el flux llegirà aquell caràcter).

9.3 Fluxos de sortida (classe `ostream`)

La classe *ostream* és una subclasse de *ios* que sobrecarrega l'operador d'inserció << per tots els tipus base i, a més a més defineix altres operacions relacionades amb l'escriptura d'informació en un flux. Aquestes operacions són les següents:

- *put*

  `stream& put(char c)`

  Escriu al flux sobre el qual es realitza l'operació el caràcter *c*.

  ```
  cout.put('w');
  cout.put(c);
  ```

  on *c* és una variable de tipus *char*.

- *write*

  `ostream& write(const char* cadena, int mida)`

  Escriu els *mida* primers caràcters de *cadena* al flux.

  Si *cadena* tingués menys caràcters que *mida*, un cop escrits els caràcters de *cadena* escriuria brossa fins completar els *mida* caràcters.

  Aquesta operació també resulta útil per generar fitxers binaris (vegeu 9.6.3).

- *flush*

  `flux.flush();`

  Força la immediata sortida al dispositiu associat al *flux* de tots els caràcters que s'hagin enviat al *flux* i encara no hagin estat escrits al dispositiu.

  Aquesta operació té sentit perquè les lectures/escriptures de/a un dispositiu es fan a través de *buffers* per tal d'optimitzar el nombre d'accessos al dispositiu. Per això pot passar que determinats caràcters enviats a un flux hagin d'esperar en algun *buffer* associat a aquell flux abans de ser físicament enviats al dispositiu.

9.4 Formats

La biblioteca d'E/S de C++ ofereix la possibilitat de controlar el format de lectura i escriptura. Per això defineix operacions com *width, precision...* i indicadors de format (*dec, hex, left...*).

Aquestes possibilitats de control de format de C++ no es presenten en aquesta versió dels apunts de C++. Us proposo que consulteu un manual de C++ per tal de conèixer-les.

9.5 Estats

Les operacions que es realitzen sobre fluxos poden generar errors. Per tal de saber en tot moment l'estat d'un flux, la classe *ios* defineix:

- Un tipus enumerat *io_state* que conté les constants (*eofbit, failbit, goodbit, badbit*) que expressen les diferents circumstàncies que es poden produir després d'executar una operació d'E/S sobre un flux.
- Una variable *state* que conté l'estat del flux en cada moment. D'aquesta variable són significatius els seus 3 bits de menor pes. Cadascun d'aquests tres bits està associat a alguna circumstància (excepció) que es pot produir després de l'execució d'una operació de L/E d'un flux (el bit de menor pes, per exemple, fa referència a que s'ha llegit la marca de final de fitxer, EOF).
- Una llista d'operacions (*eof, good, fail, bad*) per tal de consultar aquest estat i una altra operació *clear* per tal de modificar-lo explícitament (observem, doncs, que la variable *state* no la consultarem ni la modificarem directament sinó a través d'aquestes operacions).

La classe *ios* té l'aspecte següent pel que fa a la gestió dels estats d'un flux.

```
class ios{

int state;  //En realitat esta definida en una superclasse de ios.

public:

...

enum io_state{

  goodbit = 0x00.
  eofbit = 0x01.
  failbit = 0x02.
  badbit = 0x04.
};

int good const;
int eof() const;
int fail() const;
int bad() const;
void clear();
void clear(int statebits);

...
};
```

Quan s'executa una operació d'E/S sobre un flux s'actualitza (automàticament) la variable *state* (activant els bits corresponents a les excepcions detectades, si se n'ha detectat alguna) Cadascun d'aquests bits té una constant definida a l'enumerat *io_state* i una operació per consultar-la. Presentem seguidament aquestes constants i les seves operacions associades:

- *goodbit*

El valor de la constant *goodbit* és en realitat 0. Si la variable *state* té el valor *goodbit* vol dir que la darrera operació d'E/S feta sobre el flux ha estat correcta i no s'ha produït cap circumstància rellevant.

L'operació *good()* retorna *cert* si la variable *state* val *goodbit* i *fals* altrament.

- *eofbit*

  El valor d'aquesta constant és 1 (i.e. només el seu bit menys significatiu està activat).

  Si la variable *state* té el bit menys significatiu (*eofbit*) activat vol dir que s'ha fet una operació de lectura sobre el flux que ha llegit la marca de final de fitxer EOF.

  L'operació *eof()* retorna *cert* si la variable *state* té el bit menys significatiu (*eofbit*) activat i *fals* altrament. Aquesta operació la utilitzarem per tal d'acabar un procés de lectura d'un fitxer. Per exemple:

  ```
  while (!f.eof())...
  ```

- *failbit*

  El valor d'aquesta constant és 2 (i.e. només el bit de pes 2 està activat).

  Si la variable *state* té el bit de pes 2 (*failbit*) activat vol dir que s'ha fet una operació d'E/S sobre el flux que ha generat algun error però que *no ha perdut informació del flux.*

  L'operació *fail()* retorna *cert* si la variable *state* té el bit de pes 2 (*failbit*) activat o el bit de pes 4 (*badbit*) activat i *fals* altrament.

- *badbit*

  El valor d'aquesta constant és 4 (i.e. només el bit de pes 4 està activat).

  Si la variable *state* té el bit de pes 4 (*badbit*) activat vol dir que s'ha fet una operació d'E/S sobre el flux que ha generat algun error i que *s'ha perdut informació del flux.*

  L'operació *bad()* retorna *cert* si la variable *state* té el bit de pes 4 (*badbit* activat i *fals* altrament.

Finalment hi ha una operació que ens permet modificar explícitament el valor de la variable *state*: *clear.*

```
f.clear();
```

Posa a 0 la variable *state.*

```
f.clear(ios::eofbit);
```

Posa la variable *state* al valor *eofbit* (i.e. activa el seu bit menys significatiu).

Una operació d'E/S pot activar simultàniament més d'un bit de la variable *state.* Si es fan més operacions d'E/S després de l'activació d'algun d'aquests bits, aquestes s'ignoren i els bits de *state* continuen actius.

9.6 Treball amb fitxers: Les classes *ifstream, ofstream* i *fstream*

L'objectiu que ens hem proposat inicialment era el de fer operacions d'E/S de manera uniforme i independent del dispositiu. Per això hem definit un dispositiu abstracte (*el flux*) com un objecte del programa sobre el qual realitzem totes les operacions d'E/S sobre qualsevol dispositiu real. Fins ara hem realitzat operacions d'E/S amb els dispositius de l'*entrada estàndar* (usualment el teclat) i la *sortida estàndar* (usualment el monitor). Hem vist que aquests dispositius ja tenen, per defecte, un flux associat (*cin* en el cas de l'entrada estàndar i *cout* i *cerr* per la sortida estàndar).

Els fitxers també podem considerar-los com a dispositius reals sobre els quals voldrem aplicar les operacions habituals d'E/S (<<, >>, *get, getline, put,...*). La diferència amb el teclat i el monitor és que per defecte *no estan associats a cap flux del programa.*

Com les operacions d'E/S només poden aplicar-se a fluxos, per tal de poder operar amb un fitxer ens caldrà associar-lo (lligar-lo) a un flux del programa. Les clases *ifstream, ofstream* i *fstream* serveixen per això.

Aquestes classes:

1. Tenen definida una operació *open* per lligar el flux sobre el qual s'aplica a un fitxer. Aquest lligam permet fer només operacions de lectura si el flux és de la classe *ifstream*, només d'escriptura si el flux és de la classe *ofstream* i d'ambdues menes si el flux és de la classe *fstream*.

2. Tenen definida una operació *close* que deslliga un flux del fitxer al qual estava associat.

3. La classe *ifstream* hereta les operacions habituals d'entrada definides a la classe *istream*. La classe *ofstream* hereta les operacions de sortida definides a *ofstream* i la classe *fstream* hereta les d'E/S definides a *iostream*. D'aquesta manera ens assegurem que podem aplicar sobre els fitxers les mateixes operacions d'E/S que s'apliquen habitualment sobre fluxos.

9.6.1 L'operació *open*

```
flux.open(nom_fitxer, mode_accés);
```

Associa l'objecte *flux* amb el fitxer anomenat *nom_fitxer* i hi permet fer les operacions compatibles amb el mode d'accés especificat.

Els tres modes d'accés més importants són:

- *ios::in* Permet realitzar únicament operacions de lectura sobre el fitxer.
- *ios::out* Permet realitzar únicament operacions d'escriptura sobre el fitxer.
- *ios::app* Permet afegir dades a la fi del fitxer.

Podeu consultar en un manual de C++ la resta de modes d'accés (el mode *bin* es presenta a 9.6.3). Els modes d'accés es defineixen a la classe arrel de la jerarquia d'E/S (*ios*).

El mode d'accés es pot ometre en les operacions *open* sobre fluxos de les classes *ifstream* (es pren per defecte *ios::in* com a mode d'accés) i *ofstream* (en aquest cas es pren per defecte *ios::out*).

Un flux es pot lligar alternativament a un fitxer utilitzant una operació constructora de flux en lloc de l'operació *open*:

```
fstream flux(nom_fitxer, ios::out);
```

Un cop associat un flux a un fitxer és bo consultar l'estat del flux per comprovar que durant l'operació d'associació no s'han produït errors (e.g. el fitxer no exisitia...). A 9.5 es presenta com es poden fer comprovacions de l'estat d'un flux:

```
fstream f;

f.open("fitxer.dat", ios::in);

if (f.fail()) { cerr<<"error en l'apertura";
          throw (error);}
...
```

A 9.6.2 presentem exemples de treball amb fitxers.

L'operació *close*

```
flux.close();
```

Trenca la connexió entre el flux i el fitxer al qual estava associat. El flux pot utilitzar-se novament durant el programa.

És molt important acabar el treball (especialment d'escriptura) sobre un fitxer amb una operació *close* sobre el flux que té associat. Això és així perquè les operacions d'escriptura sobre un flux *no* actualitzen directament el fitxer associat al flux, sinó que es fan sobre *buffers* associats al flux. *close* provoca l'escriptura dels *buffers* al fitxer. Si no tanquem la connexió entre el flux i el fitxer podria passar que una part de la informació emmagatzemada als *buffers* associats al flux no s'hagués escrit al fitxer i es perdés.

9.6.2 Exemples de treball amb fitxers

Fitxers d'enregistraments

Un fitxer d'enregistraments és un fitxer que conté una seqüència d'enregistraments d'un cert tipus.

Les operacions d'E/S que tenen sentit xobre aquest tipus de fitxers són les que llegeixen (escriuen) un enregistrament del (al) fitxer (podme dir que la unitat de lectura/escriptura per aquests fitxers és l'enregistrament).

Suposem que volem crear un fitxer d'estudiants. Caldrà seguir els següents passos:

1. Primer de tot crearem una classe *estudiant* (si no existia) amb els atributs rellevants (que poden ser *nom, nif, edat* i *data de naixement*) i les operacions constructores, de consulta i de modificació d'aquests atributs:

```
 class estudiant{

   int edat;
   char nom[30];
   char nif[10];
   data dNaixem;
public:

  estudiant();
  estudiant(int, char*, char*, data&);
  obtNom(char*);
  modNom(char*);
  ...
};
```

Queda clar que haurà d'exisitr també una classe *data* amb una definició semblant a la següent:

```
class data{

 int dia;
 int mes;
```

```
 int any;
public:

 data(int pdia, int pmes, int pany){dia=pdia; mes=pmes; any=pany;}
 ...
};
```

2. Haurem de sobrecarregar per la classe *estudiant* els operadors >> i << que ens serviran per fer les operacions d'E/S sobre el fitxer.

```
#include <fstream>

 class estudiant{

  // ...
 public:

  estudiant();
  estudiant(int, char*, char*, data&);
   ...
  friend istream& operator>>(istream& ist, estudiant& e);
  friend ostream& operator<<(ostream& ost, estudiant& e);

};


 ostream& operator<<(ostream& ost, estudiant& e)
 {
   ost<<e.nom<<"\n"<<e.nif<<" "<<e.edat<<" "<<e.dNaixem<<"\n";

   return ost;
 }

 istream& operator>>(istream& ist, estudiant& e)
 {

   ist.get(e.nom,30,'\n');
   ist.ignore(1);
   ist>>e.nif;
   ist.ignore(1);
   ist>>e.edat;
   ist.ignore(1);
   ist>>e.dNaixem;
   ist.ignore(1);

   return ist;
 }
```

Notem que aquests operadors llegeixen (escriuen) els atributs de *estudiant*. La lectura (escriptura) de enters i cadenes de caràcters està predefinida. No així la de *dates*. Per tant, caldrà sobrecarregar a la classe *data* els operadors << i >>:

```
 class data{
  ...
 public:
 ...
  friend istream& operator>>(istream& ist, data& d);
  friend ostream& operator<<(ostream& ost, data& d);
};

 ostream& operator<<(ostream& ost, data& d)
 {
   ost<<d.dia<<" "<<d.mes<<" "<<d.any<<"\n";

   return ost;
 }


 istream& operator>>(istream& ist, data& d)
 {
   ist>>d.dia;
   ist.ignore(1);
   ist>>d.mes;
   ist.ignore(1);
   ist>>d.any;
   ist.ignore(1);

   return ist;
 }
```

3. Com << i >> només es poden utilitzar sobre fluxs, haurem d'associar un flux del programa al fitxer que volem llegir/escriure.

```
void main()
{
 fstream f;

 //ifstream f; per nomes lectura
 //ofstream f; per nomes escriptura

 f.open("nomfit.dat", ios::out); //per escriure
 f.open("nomfit.dat", ios::in); //per llegir
}
```

4. Ja podem realitzar les operacions d'E/S sobre el fitxer a través del flux al qual està lligat.

```
void main()
{
 fstream f;
 estudiant estu(...);

 f.open("nomfit.dat", ios::out);
```

```
 f<<estu;

 //o be:
 //f.open("nomfit.dat", ios::in);
 //f>>estu;
}
```

5. Al final cal dissociar el flux del fitxer.

   `f.close()`

En aquest punt és interessant fer un parell d'observacions:

- El fitxer que hem construït d'aquesta manera és un fitxer ASCII encara que tenim algun atribut dins de la classe *estudiant* (*edat*) que és binari (tipus *int*). Això és així perquè la sobrecàrrega dels operadors << i >> sobre elements de tipus *int* els escriuen (llegeixen) a (de) un flux en forma de cadenes de caràcters.
- La redefinició per cada nova classe dels operadors << i >> els converteixen en operadors d'E/S universals.

Algorisme d'escriptura d'un fitxer

```
void main()
{
 ofstream f;

 f.open("nomfit.dat", ios::out);
 obtenirDadesEstudiant(e);
 while (!final(e)){

  f<<e;
  obtenirDadesEstudiant(e);
 }
f.close();
}
```

Algorisme de lectura d'un fitxer

```
void main()
{
 ifstream f;
 estudiant e;

 f.open("nomfit.dat",ios::in);

 f>>e;
 while (!f.eof()){
  tractar(e);
  f>>e;
```

```
 }
 f.close();
}
```

Fitxers de text

Fins ara hem tractat el cas dels fitxers d'enregistraments, això és, els fitxers que es poden veure com una seqüència d'enregistraments d'una certa classe. Existeix, però, un altre tipus molt habitual de fitxers: són aquells que es poden veure simplement com una seqüència de caràcters. Podem posar com a exemple un fitxer que contingui un programa font escrit en C++. Els components que integren aquests fitxers no són pas enregistraments sinó caràcters. Per tant, les operacions de llegir elements d'aquests fitxers o d'escriure elements a aquests fitxers hauran de llegir i escriure, no enregistraments sinó caràcters.

Les operacions adients per fer això són:

- *get* i *getline* per lectura i
- *put* per escriptura.

  Un fitxer de text és un fitxer el contingut del qual s'interpreta com una seqüència de caràcters. La unitat de lectura/escriptura sobre fitxers de text és el caràcter.

Exemple 9.6.1 *El programa següent fa una còpia d'un fitxer de text.*

```
#include <fstream.h>

void main()
{

 ifstream f("text.txt",ios::in);
 ofstream f2("copia.txt",ios::out);
 char c;

 f.get(c);
 while (!f.eof()){
  f2.put(c);
  f.get(c);
 }
 f.close();
 f2.close();
}
```

9.6.3 Fitxers binaris

El *byte* és la unitat mínima d'informació que podem col.locar en un fitxer. Des d'aquest punt de vista, i adoptant una visió de baix nivell, podem considerar un fitxer com una seqüència de *bytes*. Cada *byte* d'un fitxer pot representar un caràcter expressat mitjançant el seu codi ASCII (i així tindrem un fitxer de caràcters) o alguna altra informació que no s'expressi en forma de caràcters representats amb el seu codi ASCII (així tindrem un fitxer binari). Posem dos exemples de fitxers binaris:

- Un fitxer que conté un programa executable per a un determinat processador. Els *bytes* d'aquest fitxer no representen de cap manera caràcters en codi ASCII sinó que codifiquen (en binari) instruccions del codi màquina d'aquell processador.

- Un fitxer que conté valors enters (emmagatzemats en format *int*). Habitualment un *int* ocupa dos bytes que no representen cap codi ASCII sinó que expressen en complement a 2 un determinat valor enter (e.g. el número 1026 s'expressaria en complement a 2 amb la següent configuració de bits[5]: 0000010000000010. Els primers 8 bits fan referència al *byte* més significatiu i els darrers 8 bits al menys significatiu).

  Notem que si volem representar 1026 textualment (com a cadena de caràcters), necessitarem 5 bytes: cadascun dels 4 primers representarà el codi ASCII d'un dels dígits i el darrer representarà el codi ASCII de la marca de final ('\0', en C++).

En els fitxers amb què hem treballat fins ara, cadascun dels *bytes* que contenien representava un caràcter. Això era així de forma clara en el cas dels fitxers de text però també en el cas dels fitxers d'enregistraments. Recordem, per exemple, el fitxer d'estudiants: cada enregistrament constava d'informació representada com a vectors de caràcters (el nom i el nif) i d'informació binària (l'edat i la data de naixement). Però l'operador << tradueix aquesta informació binària a cadena de caràcters i la insereix al fitxer. Així doncs, fins ara hem tractat únicament amb fitxers de caràcters.

Per fer possible les operacions de L/E sobre fitxers binaris necessitem operacions que llegeixin/escriguin un cert nombre de bytes *sense interpretar-los com a codificació ASCII de caràcters.* Les operacions de les classes *istream* i *ostream get, put, read* i *write* permeten assolir aquest objectiu.

L'única d'aquestes operacions que no hem presentat és *read*:

```
istream& read(char* info, int mida)
```

Llegeix els *mida* següents caràcters del flux i els col.loca a partir de la posició indicada per l'apuntador *info*.

Per acabar, el mode d'apertura `ios::bin` permet obrir fitxers binaris. Es poden combinar modes d'apertura fent:

```
f.open("nomfit.dat",ios::bin | ios::in);
```

9.7 La classe *FitxerSequencial*

La biblioteca estàndar d'E/S de C++ és de bastant baix nivell. Per un costat, no contempla la diferenciació dels fitxers en diversos tipus (fitxers d'accés seqüencial, d'accés directe i d'accés indexat). Cadascun d'aquests tipus hauria d'oferir un conjunt d'operacions pròpies i (en general) diferents de les dels altres tipus. Per un altre costat, el sistema de gestió dels errors que es poden produir durant la manipulació d'un fitxer és també poc refinat i res té a veure amb el sofisticat mecanisme d'excepcions que el propi llenguatge C++ defineix i encoratja.

Ens pot interessar, doncs, disposar de classes *fitxer* més abstractes que les que ens ofereix la biblioteca estàndar de C++. Classes que ens permetin diferenciar entre els diversos tipus de fitxers, associar un conjunt d'operacions molt determinades a cadascun d'aquests tipus i que ens ofereixen eines d'alt nivell per tal de gestionar els errors que es puguin produir en la gestió dels fitxers.

En aquesta secció proposo una d'aquestes classes: la *FitxerSeq* que implementa la classe dels fitxers d'accés seqüencial.

```
#include <fstream.h>

template <class T>
class FitxerSeq {
```

[5] Recordem que $1026 = 2^{10} + 2$.

```
fstream f;
int mode;

public:

FitxerSeq(){}
FitxerSeq(const char* nomfit, int tipus);

void obrir(const char* nomfit, int tipus);
void llegir(T& x);
void escriure(T& x);
void tancar();
bool fdf();

//EXCEPCIONS:

class errorApertura{};
class errorLectura{};
class errorEscriptura{};
class errorMode{};
class errorEof{};
};


template<class T>
FitxerSeq<T>::FitxerSeq(const char* nomfit, int tipus){

  mode=tipus;
  if (mode!=ios::in && mode != ios::out && mode !=ios::app)
      throw errorMode();

  f.open(nomfit,tipus);

  if (!f.good()) throw errorApertura();
}


template<class T>
void FitxerSeq<T>::obrir(const char* nomfit, int tipus){

  mode=tipus;
  if (mode!=ios::in && mode != ios::out && mode !=ios::app)
      throw errorMode();

  f.open(nomfit,tipus);

  if (!f.good()) throw errorApertura();
}

template<class T>
void FitxerSeq<T>::llegir(T& x){
```

```
if (mode==ios::out) throw errorMode();
if (f.eof()) throw errorEof();

f>>x;

if (f.fail() && !f.eof()) throw errorLectura();

}


template<class T>
void FitxerSeq<T>::escriure(T& x){

if (mode==ios::in) throw errorMode();

f<<x;

if (f.fail()) throw errorEscriptura();

}

template<class T>
bool FitxerSeq<T>::fdf()
{
  return f.eof();
}


template<class T>
void FitxerSeq<T>::tancar()
{
  f.close();
}
```

Apèndix A

Referències. Pas de paràmetres per referència

A.1 Què són les referències?

En C++ podríem definir una referència com *una manera alternativa per a referir-nos a un objecte ja existent.*

Al següent exemple presentem la sintaxi utilitzada per a definir referències:

```
int i;

int& pt=i;
```

En aquestes condicions, diem que `pt` és una referència a l'enter `i`. També podem dir que `pt` és una altra manera d'anomenar l'enter `i` o bé que `pt` és un sinònim de `i`. A la figura A.1 es presenta això mateix gràficament.

Com a conseqüència d'això:

- Només existeix un únic objecte enter, al qual es pot accedir equivalentment mitjançant `i` o `pt`.
- Treballar amb `i` o amb `pt` serà absolutament equivalent, com es mostra en l'exemple de més avall.
- La instrucció `int& pt=i` **no** és una assignació sinó la declaració de `pt` com a referència i la seva inicialització a la variable entera `i`. El vincle establert entre `pt` i `i` és permanent en tot l'àmbit de `i` i de `pt`.

Veiem tot seguit un exemple d'ús:

pt, i :

Figura A.1: Significat intuïtiu de les referències

Codi	Variable	Valor
int i=1;	r,i:	1
int& r=i;	x:	1
int x=r;		
r=2;	r,i:	2
	x:	1
r=r+1;	r,i:	3
	x:	1
r=x;	r,i:	1
	x:	1

A.2 Les referències constants

En la forma que hem vist a l'apartat anterior, una referència sempre ha d'estar associada a una variable. Mai a una constant. És a dir, el següent codi seria incorrecte:

```
int& i=1;  //ERROR!!!
```

Per a poder associar una referència a un objecte constant, podem fer-ho de la següent manera:

```
const int& i=1;
```

A partir d'aquest moment, `i` passa a ser una altra manera d'anomenar la constant `1`.

El valor de `i` no es pot modificar.

A.3 El pas de paràmetres per referència

A.3.1 Motivació

El pas de paràmetres per defecte en C i en C++ és el pas de paràmetres per valor. Aquest pas de paràmetres es caracteritza perque es fa una còpia del paràmetre actual sobre el paràmetre formal, essent tots dos, objectes diferents. Un exemple d'aquest pas de paràmetres el trobem a la funció `quadrat`:

```
int quadrat (int j)
{
  return j*j;
```

```
}

main()
{

  int k,h=3;

  k=quadrat(h);

}
```

L'objecte `h` és un objecte diferent de l'objecte `j`. Quan es fa el pas de paràmetres, el valor de l'objecte `h` (paràmetre actual) es copia sobre l'objecte `j` (paràmetre formal).

Aquest pas de paràmetres pateix dues contraindicacions:

1. Els paràmetres de sortida i d'entrada-sortida

   El pas de paràmetres per valor pot ser apropiat pels paràmetres d'entrada però no ho és pels paràmetres de sortida i entrada-sortida perquè els canvis fets per l'acció o funció sobre el paràmetre formal no afecten al paràmetre actual (ja que són, de fet, dos objectes diferents). Com a exemple, presentem l'acció `intercanvi` que és cridada per a intercanviar el valor dels dos paràmetres amb els quals es fa la crida:

   ```
   void intercanvi(int x, int z)
   {
     int aux;

     aux=x;
     x=z;
     z=aux;
   }


   main()
   {

     int i,j;

     i=1; j=2;

     intercanvi(i,j);

     //Malament. els valors de i i j no s'han intercanviat.
   }
   ```

2. El pas d'objectes *voluminosos*

   El pas de paràmetres per valor tampoc no és adequat per a fer el pas d'objectes de mida gran (en general, qualsevol estructura de dades que pot contenir força elements) per l'eficiència que es deriva de fer una còpia d'aquests objectes.

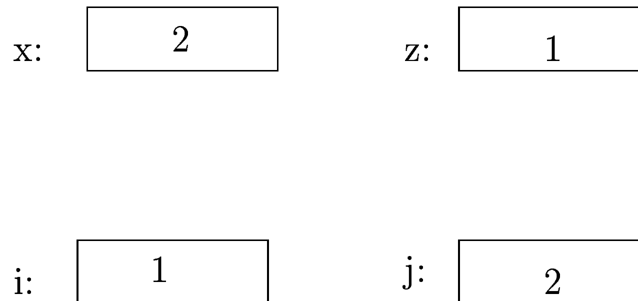


Figura A.2: Estat dels paràmetres formals i actuals després de l'execució de intercanvi.

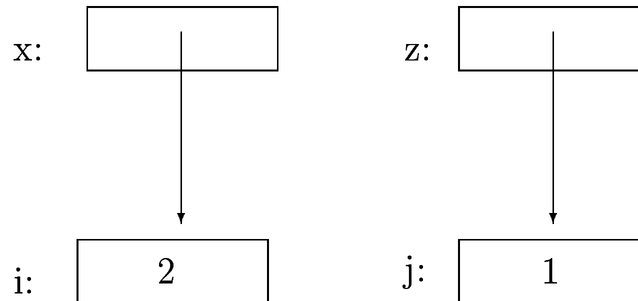


Figura A.3: Estat dels paràmetres formals i actuals després de l'execució de intercanvi2.

La forma que tenia C de resoldre aquests problemes és passar per paràmetres, no l'objecte sinó l'apuntador a l'objecte (d'això en podríem dir un pas per referència explícit). L'acció `intercanvi` es transoforma de la següent manera:

```
void intercanvi2(int* x, int* z)
{
  int aux;

  aux=*x;
  *x=*z;
  *z=aux;
}


main()
{

  int i,j;

  i=1; j=2;

  intercanvi2(&i,&j);

  //Correcte. Els valors de i i j s'han intercanviat
}
```

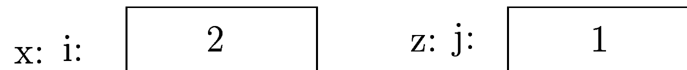


Figura A.4: Estat dels paràmetres formals i actuals després de l'execució de intercanvi3.

El bon funcionament d'aquest codi és una conseqüència del fet que ara només hi ha dos objectes: els enters `i` i `j`, mentre que els paràmetres formals `x` i `z` són apuntadors a `i` i a `j` respectivament. Els canvis de dins de l'acció es fan en realitat sobre `i` i `j`.

Aquesta forma de pas de paràmetres, però, pateix el problema de la seva poca elegància i tendència als errors. En tot moment hem de manegar explícitament els apuntadors.

C++ aporta la solució d'utilitzar les referències per al pas de paràmetres.

A.3.2 Com es fa el pas de paràmetres per referència en C++?

La idea és no passar l'objecte sinó una referència a l'objecte.

```
void intercanvi3(int& x, int& z)
{
  int aux;

  aux=x;
  x=z;
  z=aux;
}


main()
{

  int i,j;

  i=1; j=2;

  intercanvi3(i,j);

}
```

`x` i `z` passen a ser referències a `i` i `j` (o sinònims de `i` i de `j`). Per tant, les modificacions sobre `x` i `z` es fan, en realitat, sobre `i` i `j` respectivament.

El codi es conserva net i elegant ja que no hem de manejar explícitament els apuntadors.

A.3.3 Les referències constants i el pas de paràmetres

Amb el pas de paràmetres per referència que hem explicat hi ha dues coses que no podem fer:

1. Assegurar que l'acció o funció no modificarà el paràmetre passat i
2. Passar una constant. (i.e. `quadrat(2)`).

Qualsevol de les dues situacions la podem resoldre exigint que la referència que es passa per paràmetre sigui una constant. Recordem l'exemple de la funció `quadrat`:

```
int quadrat (const int& j)
{
  return j*j;
}

main()
{

  int k,k2,h=3;

  k=quadrat(h);
  k2=quadrat(2);

}
```

Ambdues crides seran ara correctes. Per altra banda, estarà prohibit de modificar el paràmetre formal `j` dins de la funció `quadrat`.

A.3.4 Quan s'ha de fer un pas de paràmetres per referència en C++?

S'ha de fer un pas de paràmetres per referència en C++ quan es compleixi alguna de les tres condicions següents:

1. Per paràmetres de sortida i d'entrada-sortida.
2. Per paràmetres (d'entrada o sortida) que tenen una mida considerable i que el seu pas per valor podria malmetre l'eficiència temporal del programa.
3. Per paràmetres polimòrfics (el tipus concret dels quals es determinarà en temps d'execució).

Apèndix B

Compilació separada

B.1 Fitxers de capcelera, fitxers d'implementació i relacions *usa*

B.1.1 Fitxers de capcelera i d'implementació

Les classes estan formades per una part d'interfície i una part d'implementació.

La part de la interfície està composada per

- La declaració dels elements (atributs i operacions) que configuren la part pública de la classe[1].
- L'especificació d'aquests elements (en particular, l'especificació de les operacions i el llistat de requeriments necessaris per l'ús correcte de la classe).

Mentre que la part de la implementació ve determinada per:

- La declaració dels elements (atributs i operacions) que formen la part privada de la classe[2]
- La implementació de totes les operacions de la classe.

La idea general de la generació en C++ d'una classe preparada per a ser utilitzada per alguna aplicació client es basa en distribuir els elements que componen aquella classe en dos fitxers:

- Un fitxer anomenat *de capcelera*, amb extensió *.h* (e.g. *tauEscacs.h*) que *aproximadament* conté la part de la interfície de la classe.
- Un altre fitxer, amb el mateix nom que el fitxer de capcelera corresponent a aquella classe però amb extensió *.cpp* (e.g. *tauEscacs.cpp*). Aquest fitxer *aproximadament* conté la part d'implementació de la classe.

Diem *aproximadament* perquè, tot i que l'esperit de C++ és el descrit, en realitat, el fitxer de capcelera, a més a més de la interfície, conté la representació de la classe.

Exemple B.1.1 *Suposem que la classe taulerEscacs està definida de la següent manera:*

[1] Sovint la part pública estarà formada només per operacions.
[2] sovint els atributs de la part privada es coneixen per la *representació de la classe*

```
enum idPeca{PEO,TORRE,CAVALL,ALFIL,DAMA,REI};
enum color{BLANC,NEGRE};


class peca{
public:

 idPeca id;
 color idcol;
};


class taulerEscacs{

 peca v[8][8];        //fileres: 0..7 i cols: 0..7
 bool enrocatBlanc;   //recordatori de si les blanques
                      //han enrocat.
 bool enrocatNegre;   //idem per les negres.

public:

 class errorInser{};

 taulerEscacs();
 void afegirPeca(int fil, int col, peca p);
 void eliminarPeca(int fil, int col);
 void mourePeca(int filOrg, int colOrg, int filDst, int colDst);
 bool amenaca(int filOrg, int colOrg, int filDst, colDst);
 //...
};

taulerEscacs::taulerEscacs()
{
//implementacio de la constructora
}
void taulerEscacs::afegirPeca(int fil, int col, peca p)
{
//Implementacio d'aquesta operacio

}

void taulerEscacs::eliminarPeca(int fil, int col)
{
//Implementacio d'aquesta operacio
}
void taulerEscacs::mourePeca(int filOrg, int colOrg, int filDst, int
colDst)
{
//Implementacio d'aquesta operacio
}
bool amenaca(int filOrg, int colOrg, int filDst, colDst)
{
```

```
//Implementacio d'aquesta operacio.
}
```

El fitxer "tauEscacs.h" corresponent a aquesta classe és el següent:

```
/***********************************************************

CLASSE  taulerEscacs


taulerEscacs();
taulerEscacs c;

Pre:
Post: t es un tauler d'escacs sense cap pe\c{c}a.
----------------------------------------------------------------
void afegirPeca(int fil, int col, peca p);
t.afegirPeca(fil,col,p)

Pre:  0<=fil,col<=7 t=t'
Post: Afegeix a t' la pe\c{c}a p a la posici\'o (fil,col). Si hi ha
      algun problema en la col.locaci\'o d'aquesta peca llen\c{c}a
      l'excepci\'o errorInser().
----------------------------------------------------------------

void eliminarPeca(int fil, int col);

Pre:...
Post:...
----------------------------------------------------------------
void mourePeca(int filOrg, int colOrg, int filDst, int colDst);

Pre:...
Post:...
----------------------------------------------------------------
bool amenaca(int filOrg, int colOrg, int filDst, colDst);

Pre: ...
Post: ...
----------------------------------------------------------------
...

enum idPeca{PEO,TORRE,CAVALL,ALFIL,DAMA,REI};
enum color{BLANC,NEGRE};


class peca{
public:

 idPeca id;
 color idcol;
};
```

```
class taulerEscacs{

 peca v[8][8];          //fileres: 0..7 i cols: 0..7
 bool enrocatBlanc;     //recordatori de si les blanques
                        //han enrocat.
 bool enrocatNegre;     //idem per les negres.

public:

 class errorInser{};

 taulerEscacs();
 void afegirPeca(int fil, int col, peca p);
 void eliminarPeca(int fil, int col);
 void mourePeca(int filOrg, int colOrg, int filDst, int colDst);
 bool amenaca(int filOrg, int colOrg, int filDst, colDst);
 //...
};

taulerEscacs::taulerEscacs()
{
//implementacio de la constructora
}
void taulerEscacs::afegirPeca(int fil, int col, peca p)
{
//Implementacio d'aquesta operacio

}
```

entre que el fitxer "tauEscacs.cpp" és:

```
#include "tauEscacs.h"

taulerEscacs::taulerEscacs()
{
//implementacio de la constructora
}
void taulerEscacs::afegirPeca(int fil, int col, peca p)
{
//Implementacio d'aquesta operacio

}

void taulerEscacs::eliminarPeca(int fil, int col)
{
//Implementacio d'aquesta operacio
}
void taulerEscacs::mourePeca(int filOrg, int colOrg, int filDst, int
colDst)
{
```

```
//Implementacio d'aquesta operacio
}
bool amenaca(int filOrg, int colOrg, int filDst, colDst)
{
//Implementacio d'aquesta operacio.
}
```

Veiem doncs, que, en realitat, el fitxer de capcelera (.h) conté:

- La part de la interfície de la classe (atributs i operacions públiques i especificació de les operacions publiques) i
- La representació de la classe (atributs i operacions privades i altres definicions de suport).

Per la seva banda, el fitxer d'implementació conté:

- La directiva d'inclusió del fitxer de capcelera (# *include*) i
- La implementació de *totes* les operacions de la classe.

Aquesta gestió utilitzada per C++ (que incorpora la representació de la classe al fitxer que conté la interfície) no és, clarament, la més idònia i elegant ni tampoc la que afavoreix més l'abstracció ja que el client de la classe, que necessita incorporar el fitxer amb la interfície, pot conèixer la representació de la classe (tot i que no hi pot accedir).

Fem un parell d'observacions addicionals respecte la distribució de les classes en els fitxers *.h* i *.cpp*:

- Sovint aquests fitxers s'utilitzen per contenir vàries classes relacionades semànticament. Per exemple, en el cas anterior, la classe *peca* (que podria ser molt més complexa amb força operacions associades) resideix al mateix parell de fitxers (.h, .cpp) que *taulerEscacs*. En general, aquesta és una pràctica habitual que facilita la descomposició d'aplicacions amb un gran volum de classes en unitats menors. És desitjable que les classes que componen cadascuna d'aquestes subunitats tinguin algun nexe d'unió (e.g. les classes que fan referència a la facturació, o a les comandes...).

  Sovint aquesta forma de descomposició es coneix amb el nom de descomposició en mòduls.

- Ja vam indicar a ??? que les classes genèriques (templates) no són pròpiament classes sinó *plantilles* que contenen les instruccions per tal de construir classes, la qual cosa la fa el compilador en el moment en què aquestes *plantilles* són instanciades (`Llista<int> l1;`. Així doncs, quan un client instancia una classe genèrica, el compilador necessita la plantilla del codi de les seves operacions per tal de crear les operacions pròpiament dites. És per aquest motiu que les classes genèriques no han de partir-se en dos fitxers (*.h* i *.cpp*) sinó que tota la classe genèrica formarà part del fitxer *.h*.

B.1.2 Relacions *usa*

Quan una aplicació client necessita utilitzar una determinada classe importa (*inclou*) la interfície d'aquella classe resident al fitxer de capcelera. Això s'aconsegueix amb la directiva de precompilació `# include`. Per exemple:

```
#include "nomFitxer.h"
```

.

La inclusió d'un fitxer *f.h* que conté la interfície d'una classe *c* dóna dret al fitxer que fa aquella inclusió (*client.cpp*) a usar les operacions i atributs definides a la part pública de *c*.

El fitxer *client.cpp* es diu que conté un programa client de la classe *c*.

L'única informació relativa a la classe *c* que es necessita per tal de compilar el fitxer *client.cpp* és la que aporta el fitxer amb la interfície *f.h*. Els codis de les funcions de *c* no són incorporades fins el procés de *link* (o *muntatge*) del programa complet (vegeu ???).

Exemple B.1.2 *En aquest exemple presentem el llistat del fitxer joc.cpp que utilitza la classe taulerEscacs que ha estat distribuïda en els fitxers tauEscacs.h i tauEscacs.cpp.*

```
#include "tauEscacs.h"


void main()
{

 taulerEscacs t;

 //Ja es poden usar les operacions
 //de la classe taulerEscacs.

 t.afegirPeca(....);
}
```

```
# include <nomFitxer.h>
```

Inclou el fitxer *nomFitxer.h* cercant-lo al directori que conté els fitxers de la biblioteca estàndar subministrada pel compilador.

```
#include "nomFitxer.h"
```

Inclou el fitxer *nomFitxer.h* cercant-lo al directori de treball i, si no hi és, al que conté els fitxers de la biblioteca estàndar subministrada pel compilador.

L'efecte exacte de la directiva `# include` és la substitució per part del precompilador d'aquesta directiva per tot el contingut del fitxer *nomFitxer.h*.

B.1.3 Compilació condicional

La inclusió de fitxers amb la directiva `# include` pot donar lloc a redefinicions d'interfícies de classes.

Exemple B.1.3 *S'afegeix a la classe taulerEscacs l'operació:*

```
void pecesTauler(color c,Llista<peca>& lp);
```

que col.loca sobre lp *la llista de peces sobre el tauler del bàndol amb color* c.

A partir d'aquest moment la classe taulerEscacs *és client de la classe genèrica Llista*<*T*> *i el fitxer* tauEscacs.h *necessitarà fer*

```
 {\tt \# include "llista.h"}.
```

Per altre costat, el programa que usa la classe taulerEscacs si vol fer servir l'operació pecesTauler *també haurà d'incloure* llista.h *i es produirà una redefinició d'aquesta classe genèrica.*

C++ permet resoldre aquesta situació mitjançant les directives de *compilació condicional*. D'aquestes directives nosaltres usarem *ifndef* i *endif*.

Com a exemple del seu ús presentem el contingut del fitxer *llista.h* que conté la definició de la classe genèrica *Llista*< *T*>:

```
#ifndef _LLISTA_H
#define _LLISTA_H

template<class T>
class Llista{

//Definicio habitual de la plantilla de classe.

};

//Codificacio habitual de les plantilles de les ops.


#endif /* _LLISTA_H  */
```

La interpretació de les directives de precompilació que apareixen al fitxer és la següent:

Si l'etiqueta `_LLISTA_H` **no** ha estat definida prèviament, es defineix tot seguit i es tenen en consideració totes les instruccions C++ que conté el fitxer fins arribar al final de la directiva condicional `endif`.

Si, contràriament, l'etiqueta `_LLISTA_H` **sí** està definida (s'ha definit prèviament amb una directiva `define`), totes les instruccions C++ i directives de precompilació que hi pogués haver fins el final de la instrucció condicional (`endif`) s'ignoren (i, per tant, no es redefineix la classe genèrica *Llista*< *T*>. Dit d'una altra manera, un segon `include` de *llista.h* quedaria sense efecte).

B.2 Procés de compilació

Procés:

1. Compilació (per separat) de tots els fitxers *.cpp*. La compilació de cadascun d'aquests fitxers crea un fitxer objecte (*.o*).

   La compilació d'un fitxer *f.cpp* no necessita el codi de les operacions que *f.cpp* utilitza i que són codificades en algun altre fitxer. Només necessita la interfície (capcelera) de les mateixes (per això n'hi ha prou amb la inclusió del fitxer que conté aquelles capceleres).

   ```
   $g++ -c cataleg.cpp        //genera cataleg.o

   $g++ -c client.cpp         //genera client.o
   ```

2. Muntatge (*link*) de tots els fitxers objectes creats en el pas anterior per tal de generar un fitxer executable.

   Aquest muntatge reuneix els codis (ja compilats) de totes les operacions codificades en els diferents fitxers d'implementació.

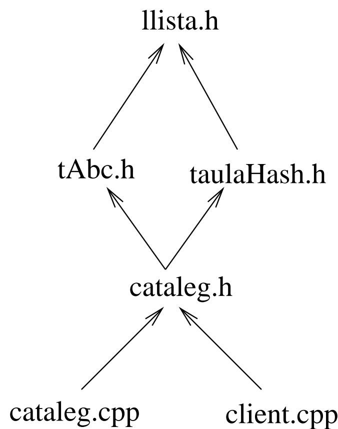


Figura B.1: Diagrama de dependències.

```
$g++ client.o cataleg.o -o client  //genera l'executable client
```

B.3 Automatització del procés de compilació

Procés:

1. Crear un fitxer anomenat *Makefile* amb el següent contingut:

```
client: cataleg.o client.o
        g++ cataleg.o client.o -o client
cataleg.o: cataleg.cpp cataleg.h taulaAbc.h taulaHash.h llista.h
        g++ -c cataleg.cpp
client.o: client.cpp cataleg.h taulaAbc.h taulaHash.h llista.h
        g++ -c client.cpp
```

   Esquema global:

   fitxer objectiu: fitxers dependència
   acció per tal de generar el fitxer objectiu

2. El procés automàtic de compilació (recompilació) s'engega fent:

```
$make client
```

   On *client* és el nom del fitxer objectiu final del proés de compilació-muntatge. .

Observacions:

- Si el fitxer *objecitu final* coincideix amb el primer objectiu del fitxer *Makefile*, no cal escriure'l en la comanda.
- El programa *make* només genera aquells fitxers necessaris per construir el fitxer objectiu tals que no estiguin actualitzats (i.e. que la seva data de creació sigui anterior a la data de creació d'algun dels fitxers dels quals depenen).

- El fitxer *Makefile* ha de contenir les instruccions i les dependències per generar tots els fitxers involucrats en un procés de compilació-muntatge.
- Es pot triar un altre nom pel fitxer *Makefile*. Si el nom escollit per aquest fitxer és *nouMake* la crida al programa make serà:

```
$make -f nouMake
```

- En general, és necessari recompilar un fitxer quan es modifica un dels fitxers dels quals depén indirectament (e.g. recompilar *cataleg.cpp* quan es modifica *llista.h*). Això és així degut a l'ús de classes genèriques i del mecanisme d'herència.