Estructures de Dades i Orientació a Objectes (EDOO)

Transparències de Teoria

Edicions UPC

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Continguts

1 Introducció: Objectius, programa i avaluació de l'assignatura 5

2 Eficiència dels algorismes 14

3 Eficiència dels algorismes: notacions assimptòtiques i regles de càlcul 36

4 TADs. Especificació d'un TAD 54

5 Implementació de TADs en C++. Constructors 74

6 Implementació de TADs en C++. Mètodes i sobrecàrrega 103

7 Implementació de TADs en C++. Operadors i sobrecàrrega 127

8 Apuntadors i gestió de memòria dinàmica 170

9 TADs: Apuntadors i gestió de memòria dinàmica 195

Continguts (cont.)

10 Estructures de Dades: Contenidors i la seva classificació. Introducció a l'STL 225

11 Estructures de Dades: Contenidors seqüencials. El contenidor vector 245

12 Estructures de Dades: Estructura de dades lista. El contenidor list 265

13 Estructures de Dades: Estructura de dades Pila. Adaptadors de l'STL. Adaptador stack 285

14 Estructures de Dades: Estructura de dades Cua. Adaptador queue 299

15 Estructures de Dades: Contenidors associatius. Contenidor Map de l'STL 315

16 Estructures de Dades: Algorismes bàsics de l'STL 331

Continguts (cont.)

17 Cas d'estudi: Programa per realitzar operacions bàsiques amb una imatge. Canvis –fer/desfer– recents d'una imatge 351

Pròleg

Aquest material cobreix el temari de l'assignatura ***EDOO***, principalment dels estudis de les enginyeries tècniques no informàtiques. S'ha procurat que sigui el més autocontingut possible. Per això, el material està estructurat per "hores". Els conceptes de teoria van acompanyats de programes per ordenador per tal de practicar individualment i independentment amb els conceptes explicats. També s'inclou un cas d'estudi perquè sigui dut a terme amb la participació dels estudiants.

Els meus agraïments a Jordi Marco pels seus comentaris i suggeriments.

Fatos Xhafa
Dept. CS / ESEIAAT, UPC

1. Introducció: Objectius, programa i avaluació de l'assignatura

- Se centra en el concepte de TADs: Definir de forma abstracta un tipus i un conjunt d'operacions
- Estructures de dades bàsiques i el seu ús correcte
- Es presenta amb programació **Orientada a Objectes en** C++

Objectius generals

Objectius derivats dels plans d'estudi de les Enginyeries Tècniques

- Adaptar els continguts al perfil d'alumnat que realitza aquests estudis
- Donar als alumnes uns coneixements generals del que representa el desenvolupament de programes a nivell industrial
- Adaptar els continguts a les necessitats de l'entorn socioeconòmic i professional del lloc geogràfic al qual per-tanyen els estudis
- Dissenyar els continguts i les metodologies tenint en compte els currículums aconsellats per les institucions nacionals/internacionals especialitzades

Objectius derivats dels plans d'estudi de GRESAUD

- Proporcionar una formació integral en els diferents àmbits d'aplicació de les tecnologies del so i la imatge
- Proporcionar les eines necessàries que permetin adquirir, processar, transmetre i emmagatzemar senyals d'audio, video, gràfics, dades, etc.
- Proporcionar els coneixements necessaris que permetin dissenyar xarxes telemàtiques per la transmissió de material audio-visual
- Proporcionar els ensenyament necessaris per entendre la creació i la realització de material audiovisual
- Proporcionar els coneixements necessaris que permetin gestionar projectes audiovisuals
- Formar tècnics capaços d'adaptar-se a les noves tecnologies durant la seva vida professional

Objectius específics de l'assignatura

- Adquirir una metodologia per especificar i implementar TADs
- Familiaritzar-se amb els conceptes bàsics de la programació *OO* en objectes
- Conèixer les estructures de dades bàsiques i aprendre a utilitzar-les correctament
- Conèixer i utilitzar els TADs genèrics de l'STL
- Disposar de criteris per escollir l'alternativa més ade-quada d'estructures de dades
- Veure, a través de les pràctiques de laboratori i del pro-jecte, l'ús dels TADs i d'estructures de dades
- Promoure l'ús de les llibreries d'estructures de dades, concretament la llibreria estàndard de l'STL

Temari de l'assignatura

Continguts teòrics

- Introducció
- Eficiència d'algoritmes
- Abstracció i Tipus Abstractes de Dades (TADs)
- Classes i objectes
- Estructures de dades (contenidors seqüencials i associatiuq

Sessions de LABORATORI

- Es publicaran per avançat els problemes/treball a fer en cada sessió

Sessions de laboratori

- Es faran servir guions/material de laboratori
- Treball en grup
- Documentació de la sessió
- Es fomentarà el lliurament de les sessions complertes i documentades.

La tutoria, semi-presencialitat i treball online en grup

- Fomentar el treball en grup online

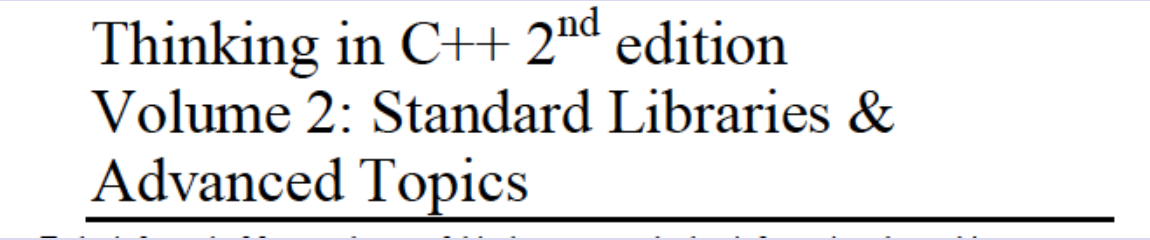
Model Guia Docent bc jÜESEIAAT, UPC.)

SISTEMA DE QUALIFICACIÓ

Examen parcial (P): 20%
Examen final (F): 30%
Controls (C1, C2): 20%
Resolució de problemes (T): 10%
Projecte (J): 20%

Per aquells estudiants que compleixin els requisits i es presentin a l'examen de re-avaluació, la qualificació de l'examen de re-avaluació substituirà les notes de tots els actes d'avaluació que siguin proves escrites presencials (controls, exàmens parcials i finals) i es mantindran les qualificacions de pràctiques, treballs, projectes i presentacions obtingudes durant el curs.
Si la nota final després de la re-avaluació és inferior a 5.0 substituirà la inicial únicament en el cas que sigui superior. Si la nota final després de la re-avaluació és superior o igual a 5.0, la nota final de l'assignatura serà aprovat 5.0.

Bibliografia bàsica



Bibliografia complementària

- Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca versió paper i digital).

2. Eficiència dels algorismes

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - **concepte de l'eficiència d'un programa**
    - eficiència temporal
    - eficiència espacial
  - **eficiència temporal d'un programa**
    - definició de l'eficiència temporal
    - factors que intervenen en l'eficiència temporal
    - mesurar l'eficiència temporal
  - **disseny de programes eficients**
- **Resum**
- **Referències**

Edicions UPC
Inici
Contingut
◀◀ ▶▶
◀ ▶
Pàgina 15
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- Eficiència és una qualitat important dels programes
- Quan un problema es pot resoldre per més d'una programa, eficiència és un criteri important per escollir quin programa usar
- Eficiència és una tècnica de disseny de programes ja que permet obtenir programes que fan un bon ús dels recursos
- Disseny d'estructures de dades eficients

Edicions UPC
Inici
Contingut
Pàgina 16
Tornar
Pantalla Completa
Tancar
Sortir

Conceptes bàsics

En dissenyar un algorisme volem que sigui:

- Correcte: qualitat fonamental i imprescindible
- Altres qualitats:
  - eficient
  - intel·ligible (clar i ben estructurat)
  - general (de fàcil ús, manteniment, etc.)

L'eficiència és considerada com una de les qualitats més importants d'un algorisme:

- Un algorisme és eficient si és *"ràpid"* i fa un *"bon ús dels recursos"* (memòria).
- L'eficiència es mesura amb dos paràmetres: *temps d'execució* (eficiència temporal) i *quantitat de recursos* que consumeix (eficiència espacial).

Edicions UPC
Inici
Contingut
Pàgina 17
Tornar
Pantalla Completa
Tancar
Sortir

Eficiència temporal

Aquest paràmetre mesura el temps de processador (d'execució del programa). S'anomena també *cost temporal*. *"Com menys temps més eficiència!"*

- Bàsicament depèn del nombre d'instruccions elementals que executa el programa i del temps d'execució d'aquestes.

Els factors que determinen l'eficiència es classifiquen en:

- Primordials:
  - *tècnica utilitzada* en dissenyar l'algorisme (ex.: recorregut *versus* cerca)
  - *"tamany de l'instància"* del problema a resoldre (ex.: sumar els elements d'una taula de llargària 10 o 10.000).
- Altres:
  - màquina en què s'executa (processador, RAM, Sistema Operatiu, . . .)
  - programari usat en la implementació (llenguatge, compilador, llibreries, . . .)

Com es mesura l'eficència d'un algorisme?

- Prescindim del factor "màquina". En canviar de màquina, compilador o llibreria... l'eficiència "s'afecta" per un factor constant. (*Teorema de la Invariància)*
- Establim l'eficiència en funció del "tamany d'entrada"/quantitat de dades sobre els quals s'executa l'algorisme
  - en general el tamany d'entrada s'expressa en termes d'un o més paràmetres (per una taula de llargària n s'expressa en funció de n, per una taula $m \times n$ s'expressa en funció de n i m)
  - anotarem per $T(n)$ la funció que expressa el temps d'execució de l'algorisme per una entrada de tamany n

L'eficiència es pot obtenir:

- mesurant el temps d'execució (prova empírica).
- fent una anàlisi matemàtica de l'algorisme (càlcul de $T(n)$)

Edicions UPC
Inici
Contingut
◀◀ ▶▶
◀ ▶
Pàgina 19
Tornar
Pantalla Completa
Tancar
Sortir

Eficiència dels algorismes: cas millor, pitjor i mitjà

L'eficiència d'un algorisme depèn també de les característiques de les dades (ex.: "*cerca d'un element a la taula*")

- L'element buscat es troba al "*principi*", al "*mig*", al "*final*" o no es troba.

A l'hora de calcular el cost temporal, distingim:

- Cas millor (poc interessant!)
- Cas pitjor (representatiu)
- Cas mitjà (sol ser difícil de calcular)

Considerarem el cas pitjor:

- Dóna més informació
- Dóna una cota superiror per totes les instàncies
- Sol ser més fàcil de calcular que el cas mitjà

Càlcul de $T(n)$

Dues maneres bàsiques són:

- Expressar $T(n)$=nombre d'instruccions *elementals* (primitives) executades per l'algorisme sobre instància de tamany n
  - assignació (denotem T_a el temps necessari per fer-la)
  - comparació (denotem T_c el temps necessari per fer-la)
  - avaluació d'operadors i funcions elementals (no són igualment costoses...)
- Expressar $T(n)$ assimptòticament. Interessa "*l'ordre de magnitud*" (per valors grans de n)
  - prescindim de les constants multiplicatives (ex.: $T(n) = 3n^2$ –ordre quadràtic)
  - prescindim de factors d'ordre de magnitud menor (ex.: $T(n) = 3n^2 + 7n + 10$ –ordre quadràtic)

Exemple de càlcul "exacte" de $T(n)$

Sumar els elements de posicions múltiples de 3 d'una taula de n elements.

```
...
S = 0; i = 0;          //Ta, 2 cops

while (i < n){         //Tc, n cops
  if (i % 3 ==0)       //Tc, n cops
     S = S + t[i];     //Ta, n/3 cops
  i = i + 1;           //Ta, n cops
}
...
```

Comptem: $T(n) = 2 \cdot Ta + 2n \cdot Tc + \lfloor \frac{n}{3} \rfloor \cdot Ta + n \cdot Ta = \left(2 + \lfloor \frac{n}{3} \rfloor + n\right) \cdot Ta + 2n \cdot Tc$

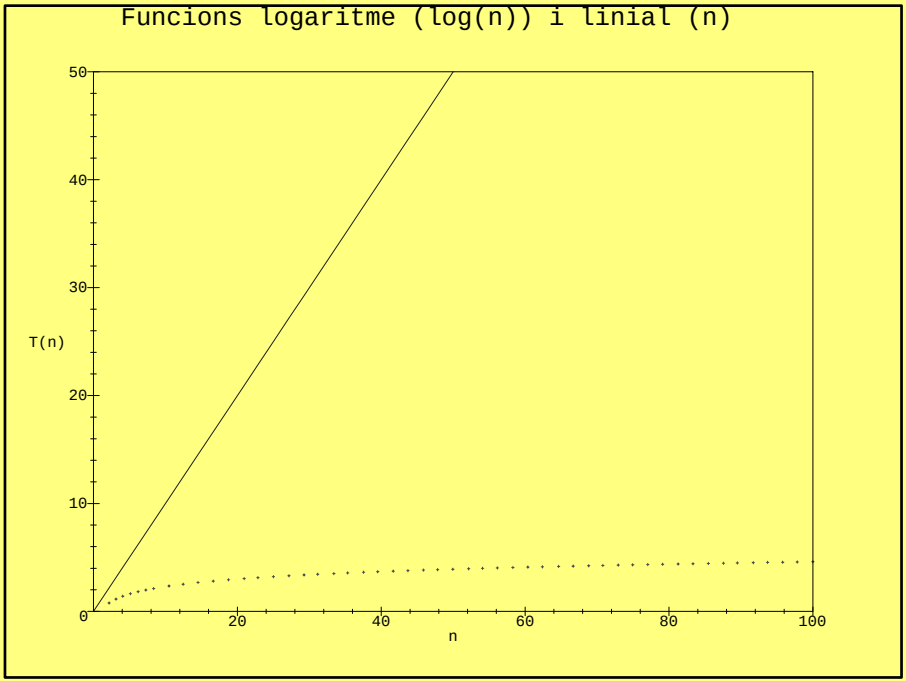
En general, és laboriós fer els càlculs. Es poden obtenir expressions de $T(n)$ complexes, no gaire llegibles, ...

Formes de creixement bàsiques

- Constant
- Logarítmica (ex.: $3\log(n) + 4$)
- Sublinial (ex.: $\sqrt{n}$)
- Linial (ex.: $7n + 5$)
- Quasi linial (ex.: $n\log(n)$)
- Quadràtica (ex.: $n^2 + 3n + 4$)
- Polinòmic (ex.: n^k)
- Exponencial (ex.: 2^n)
- Factorial (ex.: $n!$)

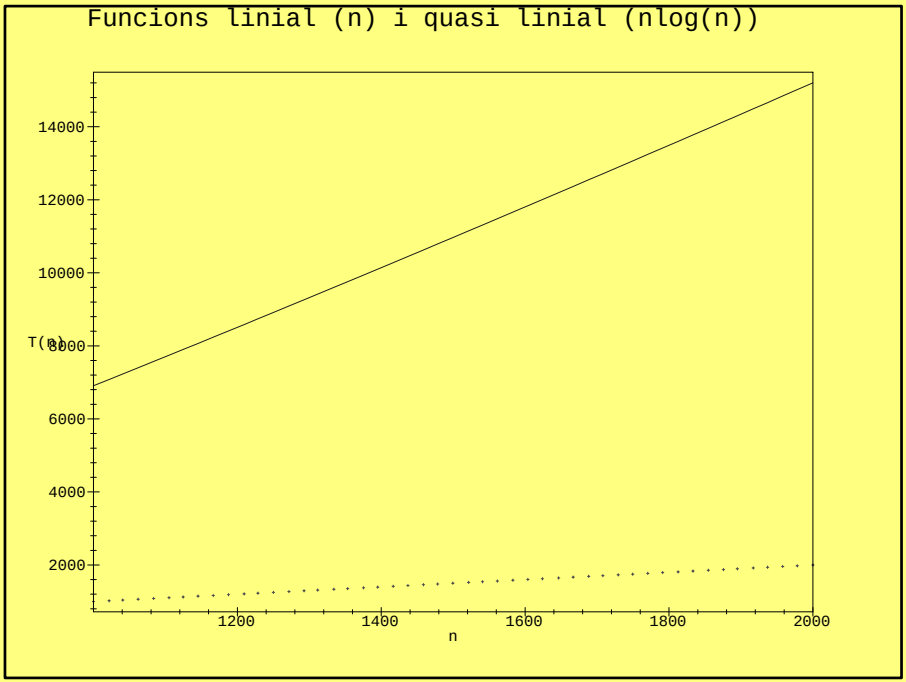
Edicions UPC
Inici
Contingut
Pàgina 22
Tornar
Pantalla Completa
Tancar
Sortir

Creixement logarítmic i linial



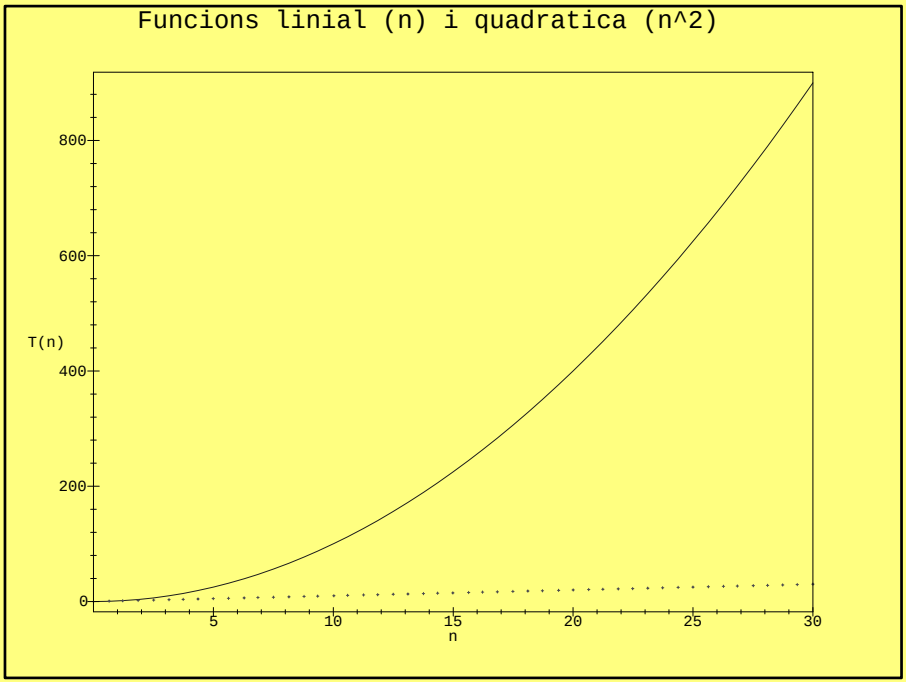
Edicions UPC
Inici
Contingut
Pàgina 23
Tornar
Pantalla Completa
Tancar
Sortir

Creixement linial i quasi-linial



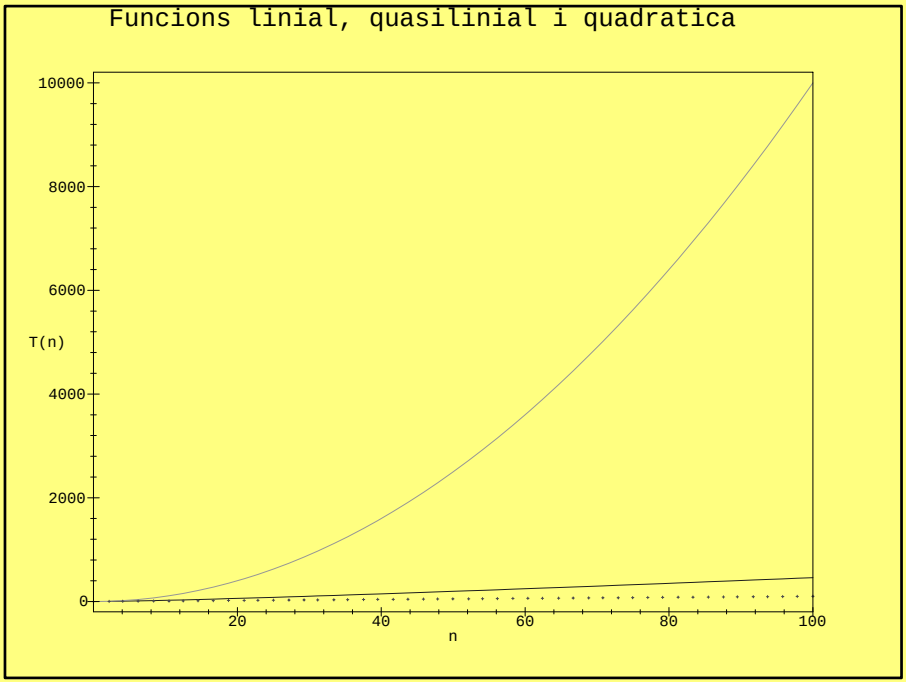
Edicions UPC
Inici
Contingut
Pàgina 24
Tornar
Pantalla Completa
Tancar
Sortir

Creixement linial i quadràtic



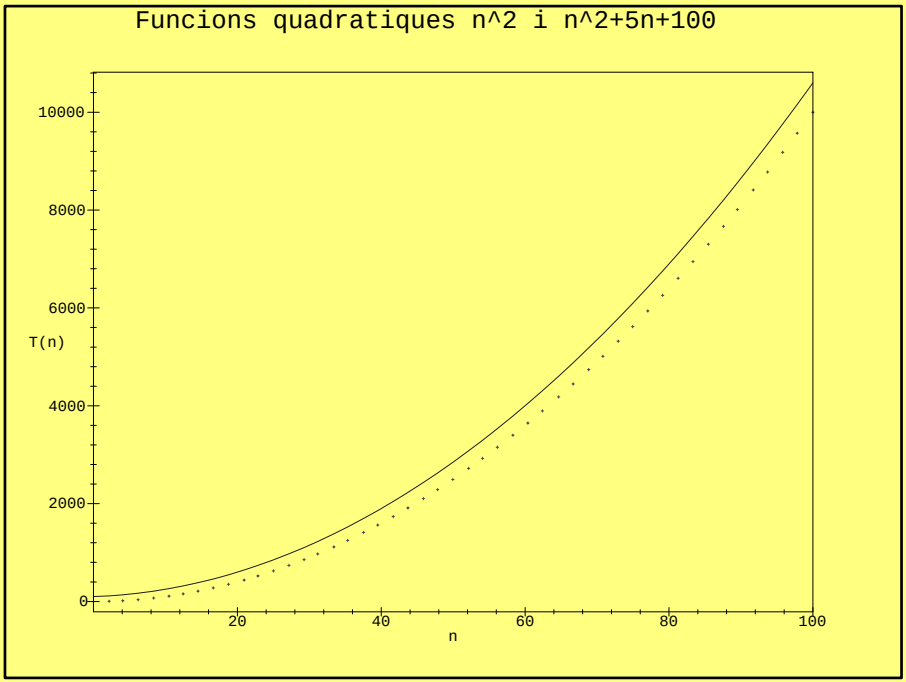
Edicions UPC
Inici
Contingut
Pàgina 25
Tornar
Pantalla Completa
Tancar
Sortir

Creixement linial, quasi-linial i quadràtic



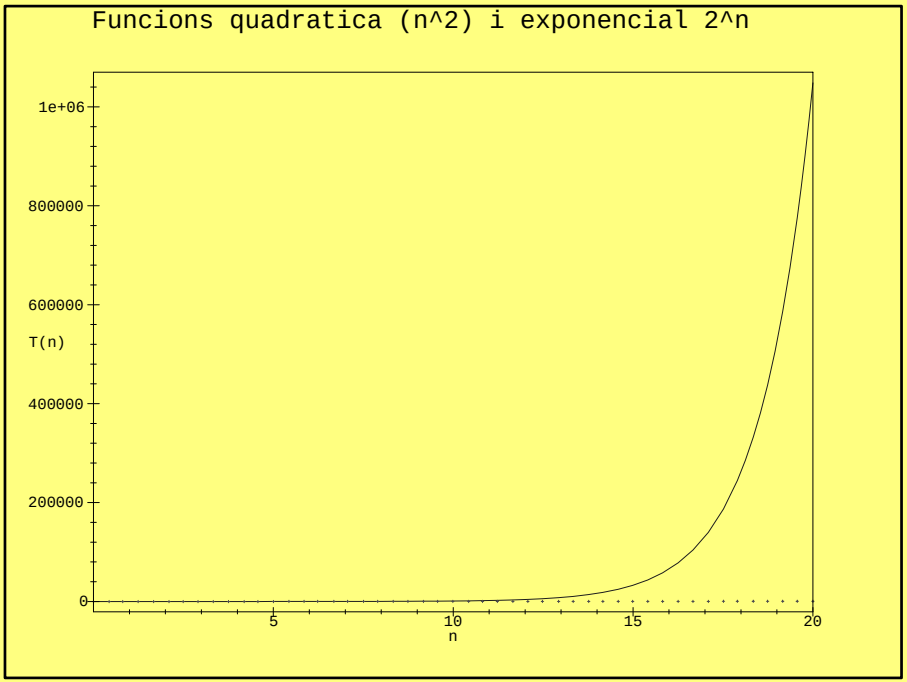
Edicions UPC
Inici
Contingut
Pàgina 26
Tornar
Pantalla Completa
Tancar
Sortir

Creixement quadràtic



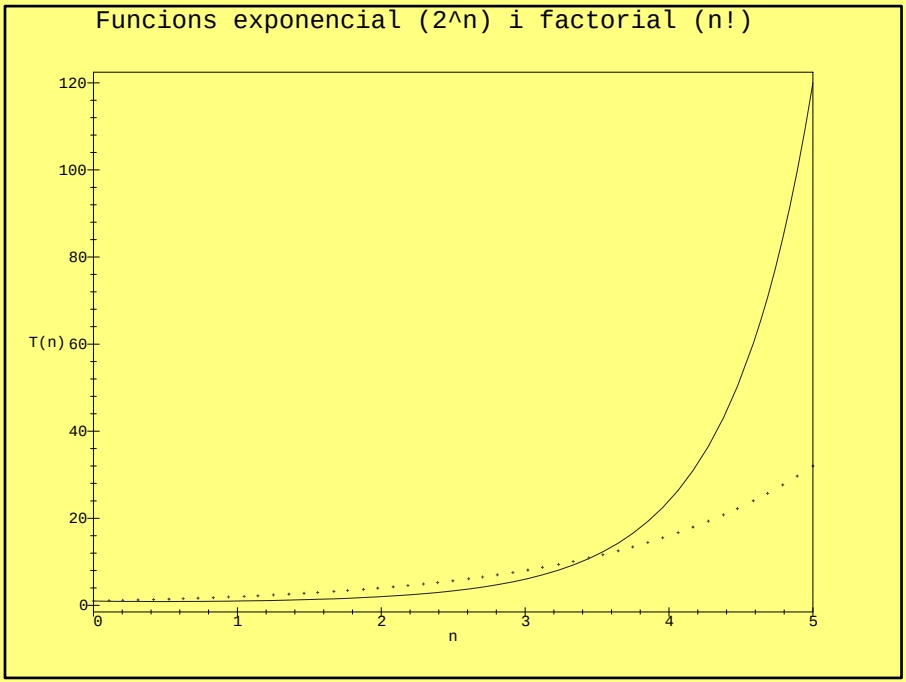
Edicions UPC
Inici
Contingut
Pàgina 27
Tornar
Pantalla Completa
Tancar
Sortir

Creixement quadràtic i exponencial



Edicions UPC
Inici
Contingut
Pàgina 28
Tornar
Pantalla Completa
Tancar
Sortir

Creixement exponencial i factorial



Edicions UPC
Inici
Contingut
Pàgina 29
Tornar
Pantalla Completa
Tancar
Sortir

Disseny d'algorismes eficients: comprovar si un nombre és primer

Un nombre natural n és *primer* si es divideix només per l'1 i ell mateix.

Versió 1: Comprovem si hi ha algun divisor de n entre $2 \ldots n-1$

```
bool esPrimer(int n){
    int i=2;
    bool trobat=false;
    while(i<=n-1 && !trobat)
      if (n%i==0) trobat=true; else i++;

     return !trobat;
}
```

- Cost temporal: en el cas pitjor $n-2$ iteracions
- *Nota*: Podem millorar la condició del bucle?

Disseny d'algorismes eficients: comprovar si un nombre és primer (cont.)

Versió 2: Els divisors de n, si existeixen, estan entre $2 \ldots n/2$

```
bool esPrimer(int n){
  int i=2;
  bool trobat=false;
  while(i<=n/2 && !trobat)
    if (n%i==0) trobat=true; else i++;

  return !trobat;
}
```

- Cost temporal: en el cas pitjor $\lfloor n/2 \rfloor - 1$ iteracions

Disseny d'algorismes eficients: comprovar si un nombre és primer (cont.)

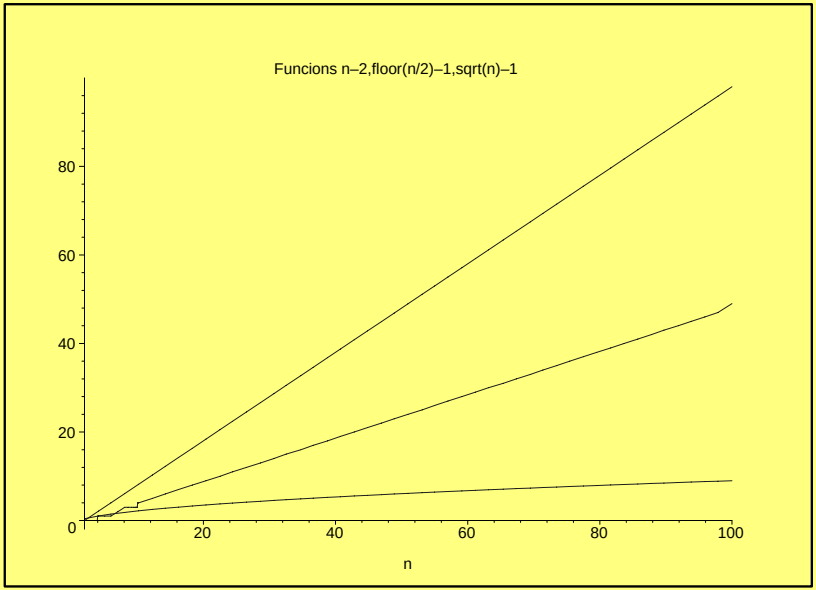
Versió 3: Si n té divisors, alguns d'ells estaran entre $2 \ldots \sqrt{n}$

```
bool esPrimer(int n){
  int i=2;
  while(i<=sqrt(n) && n%i!=0)
    i++;

  return i>sqrt(n);
}
```

- Cost temporal: en el cas pitjor $\sqrt{n} - 1$ iteracions
- Versió millorada: Comprovar si el nombre és dividit per 2. En cas que no cercar el divisor entre els 3, 5,...$\sqrt{n}$.

Comparació de la complexitat de les tres versions de la funció esPrimer



Edicions UPC
Inici
Contingut
Pàgina 33
Tornar
Pantalla Completa
Tancar
Sortir

Resum

- L'eficiència és una qualitat important dels algorismes
- L'eficiència es refereix al bon ús dels recursos (temps i espai)
- L'eficiència depèn de molts factors, dels quals la tècnica de disseny i el tamany de l'instància són els més importants
- L'eficiència es pot mesurar de diferentes maneres, l'ordre de creixement assimptòtic estableix com creix el cost temporal en funció del tamany de la instància.
- Els ordres de creixement assimptòtic importants són: constant, logarítmic, sublinial, linial, quasi-linial, quadràtic, polinòmic, exponencial i factorial
- L'eficiència és una tècnica de disseny de programes

Edicions UPC
Inici
Contingut
Pàgina 35
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - J.L. Balcázar. *Programación Metódica*. McGraw Hill 1993 (També apunts document pdf, LSI, UPC).
  - S. Roura. *Eficiència d'algorismes*. Apunts (document pdf), LSI, UPC.
- *Complementàries*:
  - N. Wirth. *Algorithms and data structures*. Prentice-Hall 1986. Traducció castellana: Prentice-Hall Hispanoamericana.

3. Eficiència dels algorismes: notacions assimptòtiques i regles de càlcul

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - notacions assimptòtiques
  - regles bàsiques de composició
  - regles per al càlcul de cost assimptòtic de programes
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 36
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- En el tema anterior hem vist el concepte de l'eficiència dels algorismes, els factors que la determinen i les maneres bàsiques de càlcul
- Una d'aquestes tècniques consisteix establir l'ordre de creixement assimptòtic del cost temporal en funció del tamany de la instància del problema
- Es tracta bàsicament d'un càlcul matemàtic del cost temporal de diferents estructures del programa per composar-los adequadament

Notacions assimptòtiques

- Ens interessa l'*ordre de creixement* de la funció del cost temporal en termes del tamany de la instància (tamany de dades d'entrada)
- *Assimptòtic*: valors grans / molts grans del tamany d'instància
- Notacions assimptòtiques (que necessitem):
  - la O majúscula
  - la Θ

Notació assimptòtica: la O majúscula

- Dóna fita superior
- $O(f)$ denota el conjunt de funcions que "*creixen, com a molt, tant com* f" (el creixement de g no supera el de f)
- Si $g \in O(f)$ llavors f és fita superior per a g: existeix una constant c i un valor per a n, a partir del qual $g(n) \leq cf(n)$
- Exemples: si $f(n) = n^2$ i $g(n) = n\sqrt{n}$ llavors $g \in O(f)$ però $f \notin O(g)$. Si $f(n) = \sqrt{3}n^2$ i $g(n) = 4n^2$ llavors $g \in O(f)$ i $f \in O(g)$

Notació assimptòtica: la Θ

- Dóna fita superior i inferior
- $\Theta(f)$ denota el conjunt de funcions que "*creixen igual com* f"(f i g són del mateix ordre de creixement)
- Si $g \in \Theta(f)$ llavors $g \in O(f)$ i $f \in O(g)$
- Exemples: si $g(n) = n^2$ i $f(n) = 10n^2 + n$ llavors $g \in \Theta(f)$

Regles bàsiques de composició

- $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$
- $O(f) \cdot O(g) = O(f \cdot g)$
- $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$
- $\Theta(f) \cdot O(g) = O(f \cdot g)$
- $\Theta(f) + O(g) = \begin{cases} \Theta(f), & g \in O(f); \\ O(g), & g \notin O(f) \end{cases}$

Regles per al càlcul de cost assimptòtic de programes

- Cost d'avaluar una acció/funció
- Cost d'avaluar una expressió
- Cost d'una assignació
- Cost d'una composició seqüencial
- Cost d'una estructura condicional
- Cost d'una estructura iterativa

Regles per al càlcul de cost assimptòtic de programes

Suposició: Les operacions bàsiques (incloent les de lectura/escriptura, conversions de valors de tipus i d'accés a les dades) es realitzen en temps constant

Cost d'avaluar una acció/funció: El màxim entre el temps d'avaluar els seus paràmetres i el d'executar el cos d'acció/funció

Cost d'avaluar una expressió:

a) Si l'expressió no conté cap crida a una funció, el temps d'avaluació és constant

b) Si l'expressió inclou una o més crides a funcions, el temps d'avaluació és el temps de la més cara de les funcions cridades

Cost d'una assignació: El temps d'executar una assignació és el temps d'avaluar l'expressió del costat dret

Edicions UPC
Inici
Contingut
Pàgina 44
Tornar
Pantalla Completa
Tancar
Sortir

Cost d'una composició seqüencial: El temps d'executar una composició seqüència d'accions és el temps de la més cara de les accions de la seqüència

Cost d'un condicional: El temps d'executar una composició condicional és de l'ordre del màxim entre el temps d'avaluar l'expressió de la condició i el temps de la branca més cara

Cost d'una estructura iterativa: El temps d'executar una estructura iterativa és la suma dels temps d'avaluar l'expressió del bucle més la d'executar cada iteració

Exemples

- Comprovar si una matriu és idempotent.
- Càlcul de C_k^n
- Buscar una resistència a la taula de resistències i actualitzar l'stock de les seves categories cas de trobar-se, o indicar que no existeix
- Càlcul de l'índex de la fila de suma màxima entre totes les files d'una matriu

Exemple: Programa per comprovar si una matriu és idempotent

Una matriu quadrada A es diu idempotent si $A^2 = A$. Esquema de cerca. PropiedadCerca: $A^2[i,j] \neq A[i][j]$.

```
const int N=...;
// Acció per omplir la matriu amb valors llegits
// pel canal d'entrada
void omplirMatr(int A[N][N]){
     for (int i=0;i<N;i++)
       for (int j=0;j<N;j++)
        cin >> A[i][j];
}
```

Exemple (cont.): definició d'accions/funcions

```
// Funció per calcular el producte de la fila f amb la columna
// c de la matriu quadrada A(N×N)
int producteFilaCol(int A[N][N],int f,int c){
    int suma=0;
    for (int k=0;k<N;k++)
        suma=suma+A[f][k]*A[k][c];
    return suma;
}
//Acció per la propietat de cerca matriu idempotent
bool propietatCerca(int A[N][N],int i, int j){
    return producteFilaCol(A,i,j)!=A[i][j];
}
```

Exemple (cont.): definició d'accions/funcions

```
//Acció per comprovar si una matriu és idempotent
bool idempotent(int A[N][N]){
    bool trobat=false;
    int i,j;
    i=0;
    while(i<N && !trobat){
            j=0;
            while(j<N && !trobat){
                if (propietatCerca(A,i,j)) trobat=true;
                else j++;
            }
            i++;
    }
    return !trobat;
}
```

Exemple (cont.): Programa principal de prova

```
int main(){
  int A[N][N];
  omplirMatr(A);
  //Crida a la funció
  if (idempotent(A)) cout << "Si";
  else cout << "No"<< endl;
  return 0;
}
```

Anàlisi d'eficiència d'algorismes recursius

- A causa de la recursivitat, es pot expressar el temps $T(n)$ que necessita l'algorisme recursiu, en termes d'ell mateix
- Nombres de Fibonacci:

```
int Fibonacci (int n){
  if (n==0) return 0;                    //cas base
  else if (n==1) return 1;               //cas base
  else return Fibonacci(n-1)+Fibonacci(n-2); //cas general
}
```

- Cost temporal: $T(n) = T(n-1) + T(n-2) + \Theta(1)$
- Obtenim així una equació recurrent, la solució de la qual ens dóna l'expressió de $T(n)$

Dos casos bàsics

Denotem a =nombre de crides recursives activades en fer-se una crida (en cas del factorial $a = 1$, en cas del Fibonacci $a = 2$) i denotem c, la constant que indica el decreiximent de les dades en passar del problema a sub-problema/es (pot ser decreiximent aritmètic o geomètric)

- **Cas 1**: El tamany dels subproblemes decreix de manera aritmètica (ex.: factorial, nombres Fibonacci). Tenim l'equació recurrent i la seva solució:

$$T(n) = aT(n-c) + \Theta(n^k) \longrightarrow T(n) = \begin{cases} \Theta(n^{k+1}) & si \quad a = 1 \\ \Theta(a^{n/c}) & si \quad a > 1 \end{cases}$$

- **Cas 2**: El tamany dels subproblemes decreix de manera geomètrica (ex.: cerca binària). Tenim l'equació recurrent i la seva solució:

$$T(n) = aT(n/c) + \Theta(n^k) \longrightarrow T(n) = \begin{cases} \Theta(n^k) & si \quad a < c^k \\ \Theta(n^k \log n) & si \quad a = c^k \\ \Theta(n^{\log_c a}) & si \quad a > c^k \end{cases}$$

- Aplicació al factorial, Fibonacci i cerca binària

Edicions UPC
Inici
Contingut
Pàgina 52
Tornar
Pantalla Completa
Tancar
Sortir

Resum

- Una de les maneres per mesurar el cost temporal dels algorismes és establir l'ordre de creixement assimptòtic
- Correspon al cas pitjor i tracta de calcular com creix el cost temporal en funció del tamany de la instància del problema. Interessa el cas d'instàncies de tamany gran o molt gran
- Per això s'usen les notacions assimptòtiques, entre elles la O i Θ
- Es calcula el cost assimptòtic per a cadascuna de les estructures bàsiques i els resultats es composen adequadament per obtenir l'ordre assimptòtic del programa

Edicions UPC
Inici
Contingut
Pàgina 53
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - J.L. Balcázar. *Programación Metódica*. McGraw Hill 1993 (També apunts document pdf, LSI, UPC).
  - S. Roura: *Eficiència d'algorismes*. Apunts (document pdf), LSI, UPC.
- *Complementàries*:
  - N. Wirth: *Algorithms and data structures*. Prentice-Hall, 1986. Traducció castellana: Prentice-Hall Hispanoamericana.

4. TADs. Especificació d'un TAD

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - abstracció. Abstracció en programació
  - mètodes bàsics d'abstracció
  - definició d'un TAD
  - procés d'obtenció d'un TAD
  - classificació de les operacions/mètodes d'un TAD
  - llenguatges de programació i TADs
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 55
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- A l'hora de dissenyar un programa apliquem la tècnica d'abstracció per identificar les funcionalitats i els tipus de dades del programa
- L'abstracció és fonamental per resoldre problemes complexos; permet obtenir programes modulars
- Els programes complexos necessiten tipus de dades i estructures de dades complexes
- Els Tipus de Dades Abstractes (TADs) són la peça fonamental de la programació modular
- L'objectiu del tema és el disseny de TADs eficients

Abstracció

En el procés del nostre coneixement sobre els fenòmens complexos, l'abstracció és l'eina més potent de què disposem...

> **abstreure**: Aillar amb la pensa, considerar separadament (un atribut o una qualitat), considerar (una part) separant-la del tot. *Abstreure l'universal del particular.*
>
> Diccionari de la llengua catalana,
> Enciclopèdia Catalana, 1994.

> *L'**abstracció** té a veure amb el coneixement de les similituds entre els objectes, situacions o procesos del món real, l'estudi d'aquestes similituds i prescindint de les diferències.*
>
> Hoare, 1972.

Abstracció (cont.)

- És un *mecanisme* base per entendre fenòmens, processos, objectes, etc. que comprenen un gran volum d'informació
- Donat un problema a resoldre, l'abstracció ens permet
  - *identificar* aquelles característiques que són les més importants –*comunes pels objectes de la classe!*
  - deixar de banda les que no ho són
- El procés d'abstracció *simplifica l'anàlisi* i solució dels problemes complexos:
  - seguint un *mètode jeràrquic*, és a dir, de nivells successius: anàlisi descendent o anàlisi ascendent
  - ex. *Anàlisi descendent*: de nivells més generals cap a nivells específics. *Programació modular*

Abstracció en programació

- Els programes són objectes molts complexos, amb un grau molt alt d'interacció i dependències entre les parts que les componen
- L'abstracció permet simplificar la complexitat dels programes
- La història de la programació demostra que el nivell d'abstracció en programació ha anat creixent: des de programació estructurada fins a programació modular (la programació amb TADs és un pas més!)

Mètodes bàsics d'abstracció

- Abstracció per parametrització
- Abstracció per especificació

Abstracció per parametrització

- S'abstreuen els paràmetres $\Rightarrow$ un conjunt potencialment infinit de possibles càlculs mitjançant una acció/funció.
- Exemple de l'intercanvi de dos valors d'un tipus T

Abstracció per especificació

- S'abstreu la tasca a resoldre mitjançant una acció/funció (*Què fa?*)
- L'especificació (pre-condició / post-condició) garanteix el resultat del procediment sense necessitat de saber els detalls de la implementació

  És l'especificació (i no el programa!) el qui realment descriu l'abstracció; el programa la implementa
- Exemple de l'intercanvi de dos valors d'un tipus T

Programació modular

- L'abstracció i especificació donen lloc a programes modulars
- Conceptuar el programa format per mòduls (un mòdul és una "unitat lògica" del programa)
- És imprescindible per a programes de mitja i gran escala (que resolen problemes complexos)
- Mòduls de dades, funcionals o de dades i funcions

Deficiències de la programació modular:

- Creació i destrucció explícita
- Encapsulació (les dades i les operacions estan separades –desacoplades)
- Fa ocultació
- Es pot produir inconsistència en les dades

Definició d'un TAD

- *TAD*= Tipus Abstracte de Dades (J. Gutag, 1974)
  - és un conjunt de valors i un conjunt d'operacions a través de les quals es poden manipular els valors.
  - es defineix mitjançant l'especificació i és independent de la manera com es representen les dades
- *Abstracte*: independència de la representació escollida per als valors del tipus
  - en general hi ha diferentes maneres de representar les dades del tipus
- Les operacions del TAD formen la *interfície* del TAD (*API* –Application Programming Interface)
  - la interfície del TAD defineix el comportament dels objectes del tipus definit pel TAD
- Un TAD es pot instanciar en *objectes* manipulables *només* a través de l'API del TAD!

Exemple: TAD Cercle

- Valors:
  - centre, radi, el color
- Operacions:
  - indicar el radi, les coordenades del centre, el color
  - indicar l'àrea, el perímetre, l'arc d'un sector del cercle

Edicions UPC
Inici
Contingut
Pàgina 63
Tornar
Pantalla Completa
Tancar
Sortir

API del TAD Cercle

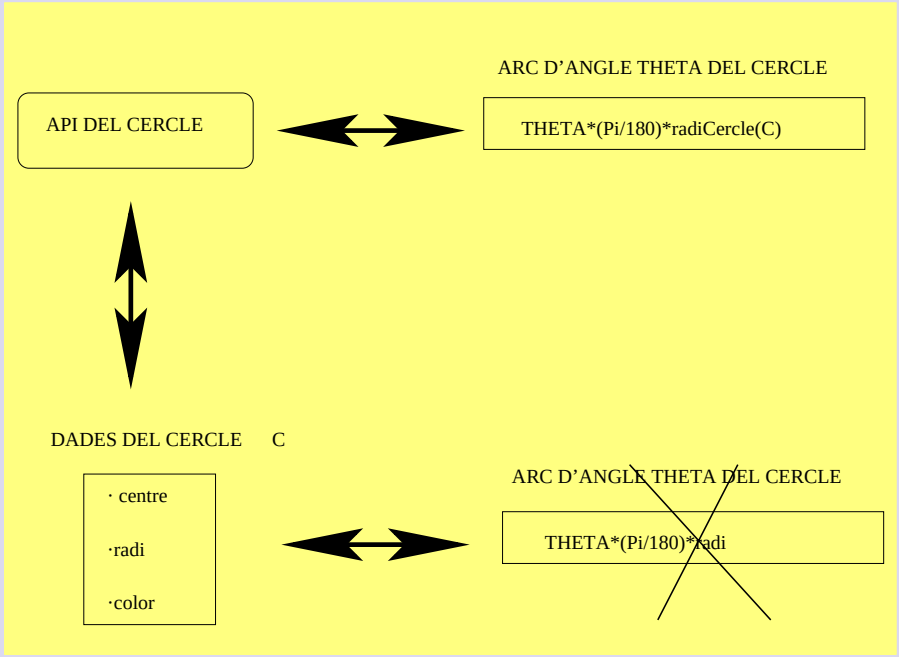
(No es fa cap suposició sobre la representació de les dades!)

```
.....
double radiCercle(Cercle C)
{
  //Pre-condició: rep un cercle C
  //Post-condició: retorna el radi de C
}
Color colorCercle(Cercle C)
{
  //Pre-condició: rep un cercle C
  //Post-condició: retorna el color del cercle C
}
double areaCercle(Cercle C)
{
  //Pre-condició: rep un cercle C
  //Post-condició: retorna l'àrea de C
}
.....
```

Edicions UPC
Inici
Contingut
Pàgina 64
Tornar
Pantalla Completa
Tancar
Sortir

Avantatge d'usar l'API del TAD

Càlcul de l'arc d'un sector de cercle d'angle Θ



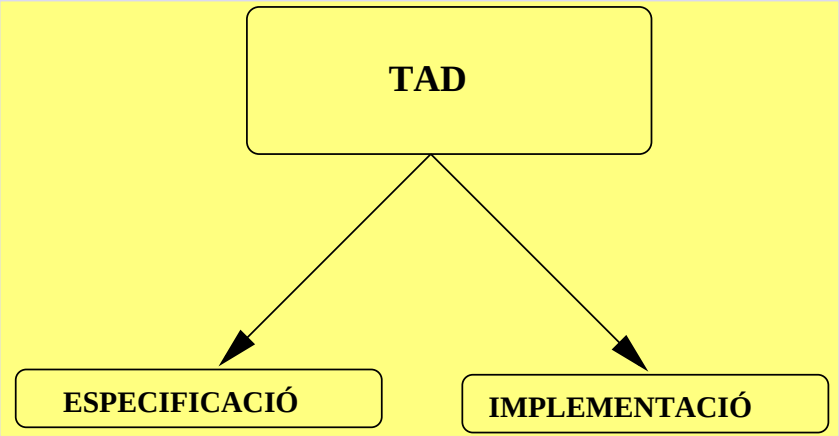
TAD Cercle: representacions

- $\{r, a, b, color\}$
- $\{a1, b1, a2, b2, color\}$ on $(a1, b1)$ i $(a2, b2)$ són dos punts de la circumferència sobre el diametre del cercle
- $\{a1, b1, a, b, color\}$ on $(a1, b1)$ és un punt de la circumferència del cercle i (a, b) coordenades del centre

L'API del Cercle s'implementa conforme la representació escollida

Procés d'obtenció d'un TAD

- Un TAD s'obté en dos pasos: *Especificació* i *Implementació*



Especificació d'un TAD: classes d'especificacions

- *Especificació formal*: Es fa usant predicats lògics (és coneguda també com especificació algebraica). Destaca per ser precisa però al mateix temps comporta un major grau de dificultat. S'expressa mitjançant la tècnica Pre-Post o a través d'equacions.
- *Especificació en llenguatge natural*: Es fa usant el llenguatge natural. Destaca per introduir ambigüetats però al mateix temps és fàcil. S'expressa mitjançant la tècnica Pre-Post.
- *Especificació quasi-formal*: Es fa usant/combinant les dues tècniques anteriors. Destaca per ser suficientment precisa i tenir un menor grau de dificultat en comparació a l'especificació formal. S'expressa mitjançant la tècnica Pre-Post.

Implementació d'un TAD

Un cop especificat el TAD, per a la seva implementació hem de completar els passos següents:

- *representació de les dades*: consisteix en escollir la representació de les dades. Val a dir que la representació en general no és única. El criteri d'elecció seria l'eficiència, és a dir, escollim aquella representació que aporta major eficiència a les operacions del TAD.
- *Implementació de les operacions del TAD*: consisteix en implementar les operacions del TAD en base de la representació escollida per a les dades.

Nota: En general els llenguatges de programació moderns permeten una separació total entre l'especificació i la implementació.

Classificació de les operacions/mètodes d'un TAD

- *Constructors primitius* (per defecte): creen objectes del TAD sense rebre cap objecte com a paràmetre.
- *Constructors*: creen objectes del TAD, reben com a paràmetre objectes d'altres tipus.
- *Constructors de còpia*: creen objectes del TAD, reben com a paràmetre objectes del mateix TAD.
- *Destructors*: eliminen l'objecte del TAD, alliberant espai de memòria.
- *Modificadors*: modifiquen els objectes del TAD.
- *Consultors*: reben com a paràmetre objectes del TAD i retornen objectes d'altres tipus.
- *Operadors*: mètodes que sobrecarreguen operadors del llenguatge
- *Iteradors*: permeten recórrer contenidors d'objectes

Llenguatges de programació i TADs

- Hi ha llenguatges de programació que no donen suport total a la programació amb TADs (ex. C)
- Els llenguatges de programació que no donen suport total a la programació amb TADs representen deficències quant a:
  - ocultació de les dades (cal ocultar les dades!)
  - inicialitació de les estructures de dades (cal inicialitzar-les!)
  - compilació separada (no s'aprofita del tot)
  - no permeten parametrització
- C++ dóna suport total a la programació amb TADs: el constructe *class* permet especificar i implementar TADs

Resum

- La programació amb TADs proporciona programes modulars i de qualitat:
  - programes llegibles
  - manteniment més senzill i menys costós
  - modificacions i extensions amb més facilitat
- TAD millora el concepte de tipus de dades:
  - *abans*: Els programes operen directament amb els valors del tipus
  - *ara*: Els programes operen amb els valors del tipus *no directament* sinó *a través de l'API del TAD*
- Benefici: en canviar la representació dels valors del TAD, *hem de tornar* a implementar les operacions del tipus *però no hem de canviar els programes*!

Criteris pràctics. Evitar errors

- S'ha d'evitar de codificar un TAD directament
- Cal separar l'especificació de la implementació
- Així, l'obtenció d'un TAD es fa en dos passos: *especificació* + *implementació*
- *Especifiquem un sol cop*, implementem tantes vegades que vulguem!
  - per cada representació de les dades $\Rightarrow$ una implementació de les operacions
- Cal codificar el TAD en un llenguatge de programació que doni suport total als TADs

Referències

- *Bàsiques*:
  - L. Joyanes. *Fundamentos de programación*. McGraw-Hill, capítol 1, 2002.
- *Complementàries*:
  - Escudero i Costa, F., Garrell i Guiu, J. M., *Fonaments de Programació*, Editorial Bruño, EUETT, capítol 5, 1993.
  - F.J. Ceballos. *C/C++ Fundamentos de programación.* RA-MA Editorial, capítol 1, 2001.

5. TADs: Implementació de TADs en C++. Constructors

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - Conceptes bàsics: Classes i objectes
  - Definició d'una classe en C++
  - Implementació d'una classe en C++
  - Constructors
  - Destructor
  - Constructors de còpia
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- Hem vist el concepte de TAD
- Hem vist els dos passos que cal seguir per obtenir un TAD: especificació i implementació
- Veurem com fer l'especificació i la implementació en C++
- La classe és la unitat de C++ que permet especificar i implementar TADs
- Veurem els diferents tipus de constructors de la classe

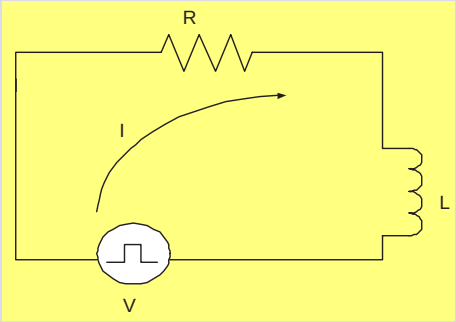
Conceptes bàsics: classes

- El constructe *class* és una extensió natual de l'*struct*
- Encapsula la informació del TAD: dades i funcions (mètodes) i proporciona visibilitat (drets d'accés)
- Assegura tots els avantatges dels TADs:
  - mantenen de l'estat dels seus objectes
  - permeten separar detalls de l'especificació dels de la implementació
  - oculten la informació
  - tenen la facilitat dels tipus predefints quant a:
    - inicialització
    - assignació
    - operacions
  - permeten fer servir operadors "usuals" a través de la sobrecàrrega

Edicions UPC
Inici
Contingut
Pàgina 77
Tornar
Pantalla Completa
Tancar
Sortir

Conceptes bàsics: objectes

- Cada objecte pertany a una classe, és instància d'una classe
- La classe és única i els seus objectes són molts
- L'objecte té estat propi: el definit pels valors dels seus atributs (membres)
- El comportament de l'objecte ve definit per la seva classe (es manipula a través de l'API de la classe)
- Exemple: Circuit RL en sèrie



Edicions UPC
Inici
Contingut
Pàgina 78
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: Circuit RL en sèrie

R - Resistència (en ohms)
L - Inductància (en henris)
V - Tensió de la font (en voltis)
f - Freqüència aplicada (en hertzs)

Es defineixen:

$X_L = 2\Pi f L$ - Reactància inductiva (en ohms)
$Z = \sqrt{R^2 + X_L^2}$ - Impedància del circuit (en ohms)

Interessa saber:

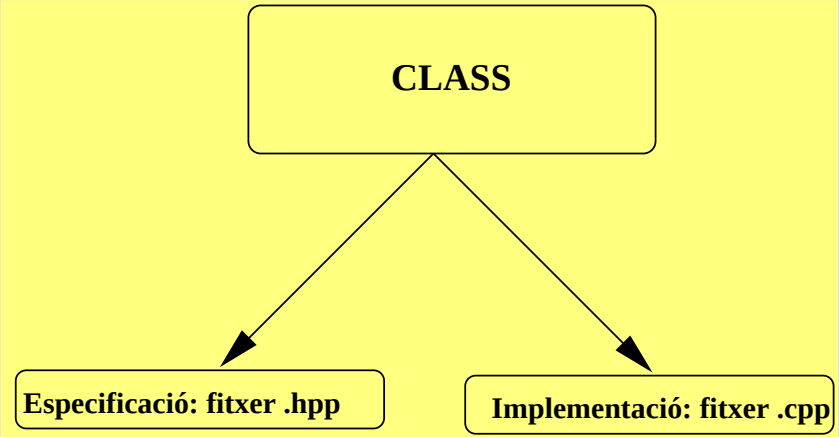
$v_L = v * X_L/Z$ - Caiguda de la tensió en la inductància (en voltis)
$v_R = v * R/Z$ - Caiguda de la tensió en la resistència (en voltis)

Exemple: Circuit RL en sèrie

- Estat de l'objecte circuit:
  - R, L, V, f
- Interfície de l'objecte circuit:
  - obtenir v_R
  - obtenir v_L
  - obtenir I
  - consultar dades del circuit

Procés d'obtenció d'una classe

- El disseny d'una classe es fa en dos pasos: *Especificació* i *Implementació*



Nota: C++ no permet separar completament l'especificació de la implementació d'una classe (a no ser que s'utilitzin classes abstractes)

Definició/especificació de classe en C++

```
class NomClasse{

private:        // Part inaccessible des de fora de la classe
                // Inclou atributs de la classe i altres
                // mètodes d'ús interns per a la classe
                // Accessibles només pels membres de la classe
                // Per defecte els atributs són privats, omissió de private
  // Declaració ATRIBUTS
  // Especificació i declaració MÈTODES

public:  // Part visible de la classe. Conté l'API de la classe
  NomClasse(params); // Constructors de la classe
                     // Mateix nom que la classe
                     // N'hi poden haver diversos
  ~NomClasse  ();// Destructor de la classe, mateix nom que la classe
  // Especificació i declaració MÈTODES
  // Declaració OPERADORS

};
```

Implementació de classe en C++

- S'implementen el/s constructor/s, destructor, mètodes i operadors conforme la representació escollida per a les dades
- La implementació és dependent de la representació
- L'especificació d'un TAD es pot implementar de diverses maneres

Operador d'accés als membres

- L'operador d'accés és el mateix que en les tuples, l'operador **.**
- Per tots els membres que no són *static* (tant membres –dades– com mètodes) s'accedeix als membres dels objectes no a la classe
- Sintaxi: nomObj.membre
- Cal respectar la visibilitat (privada o pública) dels membres

Classe en C++: membres

Aquests dos constructes són equivalents:

```
struct CircuitRL_S{
 private:  //Atributs de la classe privats
   double R; //Resistència
   double L; //Impedància
   double V; //Tensió
   double f; //Freqüència aplicada
};
```

```
class CircuitRL_C{
  //Atributs de la classe privats per defecte
   double R; //Resistència
   double L; //Impedància
   double V; //Tensió
   double f; //Freqüència aplicada
};
```

Programa de prova

```
 int main()
{
//Error: double CircuitRL_S::V is private
      CircuitRL_S C1; C1.V=120;
//Error: double CircuitRL_C::V is private
      CircuitRL_C C2; C2.V=120;
      return 0;
}
```

Classe en C++: membres static

- Els membres (dades) *static* constitueixen un mecanisme per crear membres que es comparteixen per tots ls objectes de la classe sense pertanyer a cap d'ells
- Analogia amb les variables globals; permeten evitar el seu ús

Àmbit de classe

- Una classe defineix un àmbit
- Els mètodes es declaren com funcions dins de les classes
- La definició d'un mètode pot usar membres i mètodes del mateix objecte *sense referència explícita a l'objecte en qüestió*

Implementació de classe: operador d'àmbit

- L'operador **::** és l'operador de resolució d'àmbit dels membres d'una classe ("pertànyer a")
- És necessari per a la implementació dels mètodes i accés als membres estàtics
- Sintaxi: nomClasse::nomMembre

Punter this

- Tots els membres (no *static*) tenen accés a un paràmetre implícit anomenat *this*.
  - és una variable predefinida en tots els membres d'una clase
  - conté la direcció de l'objecte concret de la classe sobre el qual s'està aplicant el membre
  - és un argument implícit
- Especialment útil pels constructors de còpia, operador d'assignació
- *this* apunta a l'objecte sobre el qual s'aplica el mètode (això explica perquè els mètodes declarats *static* no tenen *this*)

Constructors

- És un "mètode" especial: per construir objectes de la classe

  Els constructors es declaren com mètodes que *no retornen valors* amb *el mateix nom que el de la classe*

- Per a un objecte de la classe, *només s'aplica una sola vegada*!
- En crear un objecte de la classe, el compilador crida el constructor, el qual *inicialitza* l'objecte (alguns o tots els membres de la classe amb valors)
- *Tota classe ha de tenir un constructor!* Si no s'ha definit cap constructor, el compilador n'ofereix un d'"ofici" (constructor sense arguments, inicialitza els camps a zero o als seus valors per defecte)

Constructors (cont.)

- La classe pot tenir un o més constructors, el sense paràmetres s'anomena *constructor per defecte*
- És una bona pràctica definir sempre el constructor per defecte per controlar/personalitzar la creació/inicialització dels objectes de la classe.
- A través del constructor es fa reserva de memòria (cas que sigui necessari pels seus camps) que després s'alliberarà pel *destructor*

Contructors: definició

```
// Definició de la clase CircuitRL, fitxer  CircuitRL.hpp
class CircuitRL{
     //Atributs de la classe. Són privats
     double R; //Resistència
     double L; //Impedància
     double V; //Tensió
     double f; //Freqüència aplicada

     public:
     //Constructors
     CircuitRL(); //Constructor per defecte
     //Constructor amb dos paràmetres
   CircuitRL(double RR, double VV);
      // Constructor amb quatre paràmetres
     CircuitRL(double RR, double LL, double VV, double ff);
     //Resta de membres
};
```

Constructors: inicialització d'objectes

```
int main(){

//Constructor per defecte. Inicialitza R, L, V, f a 0
   CircuitRL C1;

//Constructor amb dos paràmetres.
//Inicialitza R, V als valors especificats
   CircuitRL C2(500,125);

//Constructor amb quatre paràmetres.
//Inicialitza R, L, V i f als valors especificats
   CircuitRL C3(500,1e-3,125,5e3);

   return 0;
}
```

Constructors: implementació

Per inicialitzar membres: ús d'*assignacions* o de la *llista d'inicialitzacions* (útil especialment per als membres de la classe constants o referències)

```
CircuitRL::CircuitRL(){
     R=L=V=f=0.0;
}
CircuitRL::CircuitRL(double RR, double VV){
      R=RR; V=VV;
}
CircuitRL::CircuitRL(double RR, double LL, double VV, double ff){
      R=RR; L=LL; V=VV; f=ff;
}
//Ús de la llista d'inicialitzacions
CircuitRL::CircuitRL(double RR, double LL, double VV, double ff):
R(RR),L(LL),V(VV),f(ff)
{
}
```

Constructors: implementació, llista inicialitzacions

```
class C{
      int x;
      const int c;
      int& Rx;
      //...
    public:
      C(int i):x(i),c(17),Rx(x)
      {
      }
    //...
};

int main(){
      C c(10);
      //...
      return 0;
}
```

Destructor

- Els destructors proporcionen un *mecanisme per garantir la destrucció dels objectes*
- *Només hi ha un destructor per a una clase!*
- Quan un objecte cau en desús o es demana explícitament destruir-lo, es crida automàticament el destructor:
  - en acabar l'àmbit *local* de l'objecte
  - en acabar l'execució del **main** pels d'àmbit global (o d'arxiu)
  - en invocar-se l'operador **delete** sobre un objecte

Destructor (cont.)

- Un destructor té *el mateix nom que la classe*, precedit pel caràcter $\sim$. *No admet paràmetres ni retorna cap valor!*
- La seva implementació té sentit quan en el constructor hem reservat memòria que cal alliberar
- El compilador crea un per defecte, en cas que no s'hagi proporcionat.

Exemple: Destructor

```
// Definició de la clase CircuitRL, fitxer  CircuitRL.hpp
class CircuitRL{
    //Atributs de la classe
    //...
    public:
    //Constructors
    //...

    //Destruuctor
    ~CircuitRL();
    //Resta de membres
};
// Implementació de la clase CircuitRL, fitxer  CircuitRL.cpp
//Destructor
CircuitRL::~CircuitRL(){}
```

Constructor de còpia

- Un constructor de còpia inicialitza objectes amb còpies d'objectes del mateix tipus.
- Sintaxi de *declaració*: NomClasse(const NomClasse& C), crea un objecte còpia de l'objecte C

```
//Declaració constructor de còpia
CircuitRL(const CircuitRL& C);
```

- *Implementació*:

```
//Constructor de còpia
CircuitRL::CircuitRL(const CircuitRL& C){
    R=C.R;
    L=C.L;
    V=C.V;
    f=C.f;
}
```

- *Invocació*:

```
CircuitRL C4(C3);//C4 és còpia de C3 creat prèviament
```

Constructor de còpia: observacions

- El constructor de còpia no cal si el que es vol és copiar un a un els membres (dades). El compilador ja en proporciona un per defecte
- En el cas de la classe CircuitRL no feia falta
- És necessari quan l'objecte necessita memòria dinàmica (es veurà més endavant)

El constructor de còpia s'usa a més a més:

- Quan una funció rep un objecte per valor
- Quan una funció retorna un objecte
- En les estructures de dades, quan es crida el constructor de còpia

Resum

- La classe és la unitat bàsica de la programació orientada a objectes. (La programació que tractem s'anomena programació basada en objectes ja que no es contemplen característiques d'orientació a objectes com herència i polimorfisme).
- Conceptualment, una classe es pot dividir en dues parts, segons el tipus de visibilitat (d'accés): part privada i part pública.
- Els objectes pertanyen a una classe. La classe és única i els seus objectes són molts
- Els objectes s'han de construir a través dels constructors de la classe
- Els constructors de la classe solen ser: el per defecte, el(s) amb paràmetre(s) i el de còpia
- El destructor de la classe serveix per destruir els objectes de la classe quan aquests cauen en desús

Criteris pràctics. Evitar errors

- Accedir als membres d'un objecte a través de l'API de la classe
- Els membres d'una classe no es poden inicialitzar en declarar-los (com ho fem amb les variables)!
- Els constructors i destructor no retornen cap valor
- El constructor/destructor no es una funció **void**.
- El compilador només crea un constructor d'"ofici" si no s'ha definit cap constructor, però la inicialització no es fa a valors previsbles!
- El constructor per defecte es crida NomClasse NomObj; (sense ()).

Criteris pràctics. Evitar errors (cont.)

- Els mètodes *static* no tenen un *this*
- El constructor de còpia rep com a paràmetre una referència d'un objecte de la classe
- És bo implementar sempre el constructor de còpia
- Els constructors i destructor no es poden declarar *static*
- S'ha de posar **;** després de la clau que tanca la classe

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. capítol 13, McGraw-Hill, 2002.
- *Complementàries*:
  - Walter Savitch. *Resolución de problemas con C++*. Capítols 6 i 11, Prentice-Hall, 2000.

6. TADs: Implementació de TADs en C++. Mètodes i sobrecàrrega

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - mètodes d'una classe
  - mètodes privats i públics
  - mètodes **const**
  - sobrecàrrega dels mètodes
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- En el tema anterior hem vist la sintaxi de la classe
- En aquesta sessió veurem com especificar i implementar els mètodes d'una classe
- C++ permet especificar que un mètode és només de consulta per tal de garantir que no es canvia l'estat de l'objecte de la classe
- A vegades podem necessitar implementar un mètode que fa la mateixa tasca però amb dades diferents
- Veurem el mecanisme de la sobrecàrrega que permet implementar el mateix mètode vàries vegades (amb llistes de paràmetres diferents)

Edicions UPC
Inici
Contingut
Pàgina 105
Tornar
Pantalla Completa
Tancar
Sortir

Mètodes d'una classe

- Una classe encapsula dades i funcions/mètodes
- Els mètodes de la classe serveixen per consultar i/o modificar l'estat dels objectes de la classe
- Els mètodes de la classe poden ser *privats* o *públics*:
  - els mètodes privats són d'ús intern per a la classe
  - els mètodes públics formen la interfície (API) de la classe
- Els mètodes s'han d'especificar (en llenguatge natural o quasi-formal)
- En implementar un mètode de la classe, podem usar membres i mètodes de la mateixa sense referència explícita als membres.
- Els mètodes es poden declarar *const*, no modifiquen l'estat de l'objecte
- Els mètodes es poden declarar *static*, no s'associen a cap objecte (còpia única) –no ho veurem!
- Els mètodes es poden sobrecarregar

Exemple: Classe Pixel, especificació

```
// Classe Pixel.hpp. Representació basada en el model RGB
// La directiva del preprocessador assegura la inclusió una sola vegada
#ifndef PIXEL_HPP
#define PIXEL_HPP
class Pixel{
  public:
    // Constructor per defecte
    //PRE: Cap
    //POST: Crea un pixel negre (R=0, G=0, B=0)
    Pixel();
    // Constructor amb paràmetres
    //{PRE: Rep tres reals RR, GG, BB }
    //{POST: Crea un pixel amb valors R, G, B inclosos en [0.0...1.0]
    // Si 0.0<=RR<=1.0, R=RR; Si RR<0.0, R=0.0; Si RR>1.0, R=1.0
    // Si 0.0<=GG<=1.0, G=GG; Si GG<0.0, G=0.0; Si GG>1.0, G=1.0
    // Si 0.0<=BB<=1.0, B=BB; Si BB<0.0, B=0.0; Si BB>1.0, B=1.0}
    Pixel(double RR, double GG, double BB);
```

Exemple: Classe Pixel, especificació (cont.)

```
// Destructor per defecte
~Pixel();
// Mètodes consultors
//Mètodes per consultar els valors de R, G, B del pixel

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color vermell}
double getRed();

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color verd}
double getGreen();

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color blau}
double getBlue();
```

Exemple: Classe Pixel, especificació (cont.)

```
// Mètodes modificadors
//Mètodes per modificar els valors de R, G, B del pixel

//{PRE: Rep un real RR per al nou valor del color vermell}
//{POST: Si 0.0<=RR<=1.0, R=RR; Si RR<0.0, R=0.0; sinó R=1.0}
void setRColor(double RR);

//{PRE: Rep un real GG per al nou valor del color verd}
//{POST: Si 0.0<=GG<=1.0, G=GG; Si GG<0.0, G=0.0; sinó G=1.0}
void setGColor(double GG);

//{PRE: Rep un real BB per al nou valor del color blau}
//{POST: Si 0.0<=BB<=1.0, B=BB; Si BB<0.0, B=0.0; sinó B=1.0}
void setBColor(double BB);
```

Exemple: Classe Pixel, especificació (cont.)

```
  private:
    // Atributs: valors corresponents als colors
    double R, G, B;
    // Mètode privat per ajustar els valors de R, G, B en [0.0 ... 1.0]
    //{PRE: Cap}
    //{POST: Si R<0.0, R=0.0; Si R>1.0, R=1.0
    // Si G<0.0, G=0.0; Si G>1.0, G=1.0
    // Si B<0.0, B=0.0; Si B>1.0, B=1.0}
    void ajustarValors();
};
#endif
```

Observació sobre l'ús del mètode privat

- El mètode privat ajustarValors s'invocarà pel constructor per tal d'ajustar els valors del pixel abans de crear l'objecte de la classe.

Exemple: Classe Pixel, implementació

```
// Classe Pixel.cpp
#include<Pixel.hpp>

   // Constructor per defecte
   //PRE: Cap
   //POST: Crea un pixel negre (R=0, G=0, B=0)
   Pixel::Pixel():R(0),G(0),B(0)
   {
   }

   // Constructor amb paràmetres
   //{PRE: Rep tres reals RR, GG, BB. }
   //{POST: Crea un pixel amb valors R, G, B inclosos en [0.0...1.0]
   // Si 0.0<=RR<=1.0, R=RR; Si RR<0.0, R=0.0; Si RR>1.0, R=1.0
   // Si 0.0<=GG<=1.0, G=GG; Si GG<0.0, G=0.0; Si GG>1.0, G=1.0
   // Si 0.0<=BB<=1.0, B=BB; Si BB<0.0, B=0.0; Si BB>1.0, B=1.0}
   Pixel::Pixel(double RR, double GG, double BB){
    R=RR; G=GG;B=BB;
    ajustarValors();
   }
```

Exemple: Classe Pixel, implementació (cont.)

```
 // Destructor.  No fa res ja que no es fa reserva de memòria
Pixel::~Pixel(){}
// Mètodes consultors
//Mètodes per consultar els valors de R, G, B del pixel

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color vermell}
double Pixel::getRed(){return R;}

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color verd}
double Pixel::getGreen(){return G;}

//{PRE: Cap}
//{POST: Retorna el valor corresponent al color blau}
double Pixel::getBlue(){return B;}

// Mètodes modificadors
//Mètodes per modificar els valors de R, G, B del pixel
```

Exemple: Classe Pixel, implementació (cont.)

```
//{PRE: Rep un real RR per al nou valor del color vermell}
//{POST: Si 0.0<=RR<=1.0, R=RR; Si RR<0.0, R=0.0; sino R=1.0}
void Pixel::setRColor(double RR){
 if (RR<0.0) R=0.0; else if (RR>1.0) R=1.0; else R=RR;
}
//{PRE: Rep un real GG per al nou valor del color verd}
//{POST: Si 0.0<=GG<=1.0, G=GG; Si GG<0.0, G=0.0; sino G=1.0}
void Pixel::setGColor(double GG){
  if (GG<0.0) G=0.0; else if (GG>1.0) G=1.0; else G=GG;
}

//{PRE: Rep un real BB per al nou valor del color blau}
//{POST: Si 0.0<=BB<=1.0, B=BB; Si BB<0.0, B=0.0; sino B=1.0}
void Pixel::setBColor(double BB){
  if (BB<0.0) B=0.0; else if (BB>1.0) B=1.0; else B=BB;
}
```

Exemple: Classe Pixel, implementació (cont.)

```
// Mètode privat per ajustar els valors de R, G, B en el rang [0.0 ... 1.0]
//{PRE: Cap}
//{POST: Si R<0.0, R=0.0; Si R>1.0, R=1.0
// Si G<0.0, G=0.0; Si G>1.0, G=1.0
// Si B<0.0, B=0.0; Si B>1.0, B=1.0.}
void Pixel::ajustarValors(){
  if (R<0.0) R=0.0; else if (R>1.0) R=1.0;
  if (G<0.0) G=0.0; else if (G>1.0) G=1.0;
  if (B<0.0) B=0.0; else if (B>1.0) B=1.0;
}
```

Exemple: Classe Pixel, ús

```
int main(){
//Construcció d'objectes tipus Pixel. Invocació dels mètodes consultors
  Pixel P;
  cout << P.getRed() << ' '<< P.getGreen() << ' '<< P.getBlue();

  Pixel P1(0.4, 1.3,-1.8);
  cout<<P1.getRed()<<' '<<P1.getGreen()<<' '<<P1.getBlue();

  Pixel P2(0.8, 1.0,0.0);
  cout<<P2.getRed()<<' '<< P2.getGreen()<<' '<< P2.getBlue();

  //Invocació dels mètodes  modificadors
  P.setRColor(P1.getRed());
  P.setBColor(0.7);
  cout << P.getRed() << ' '<< P.getGreen() << ' '<< P.getBlue();

  return 0;
}
```

Mètodes const

- En el cas de mètodes consultors, és bo declarar els mètodes constants
- Garanteix que el compilador detecti modificacions per error ja que els mètodes constants no poden modificar els membres de l'objecte al qual fan referència
- Sintaxi:

```
<tipus> nomMetode(params) const;
<tipus> nomClasse::nomMetode(params)const
{//implementació del mètode}
```

- Exemple:

```
double getRed() const;
double Pixel::getRed()const {return R;}
```

Sobrecàrrega de mètodes

- A vegades podem necessitar resoldre la mateixa tasca amb dades de diferents tipus
- En una classe `Triangle` podem calcular l'àrea del triangle de diverses maneres, p.ex. usant les longituds de les arestes o fórmules trigonomètriques
- La solució: Usar el mecanisme de la *sobrecàrrega* suportada per C++
- Sobrecàrrega: Definir multiples mètodes amb el *mateix nom* però amb *distintes llistes de paràmetres*
  - Dues llistes es consideren distintes si difereixen en nombre de paràmetres, o en cas de tenir el mateix nombre, difereixen en tipus de paràmetres (per posició) –en almenys un d'ells.

```
int f(int a, int b, int c);
int f(int a, int b, double d);
```

- No es pot sobrecarrgar per tipus de valor retornat!

Exemple: Classe Triangle, especificació

```
// Classe Triangle.hpp
// Definit en base de dos costats  a,b i l'angle C comprès entre ells

// La directiva del preprocessador assegura la inclusió una sola vegada
#ifndef TRIANGLE_HPP
#define TRIANGLE_HPP

#include<...>
class Triangle{

    //Atributs
    double a,b; //costats
    double C;   //angle comprès entre els costats

  public:

  //Constructor
  //{PRE: Rep els costats aa, bb i angle CC en radian}
  //{POST: Construeix el triangle definit per aa, bb i CC}
  Triangle(double aa, double bb, double CC);
```

Exemple: Classe Triangle, especificació (cont.)

```
//Constructor de còpia
//{PRE: Rep un objecte Triangle}
//{POST: Crea un objecte Triangle còpia de l'objecte donat}
Triangle(const Triangle& T);
//Destructor per defecte
~Triangle();

//Mètodes consultors
double costat_a() const;
double costat_b() const;
double costat_c() const;
double angleA() const;
double angleB() const;
double angleC() const;
//{PRE: Rep els costats a, b i c del triangle}
//{POST: Retorna el perímetre del triangle}
double perimetre() const;
//{PRE: Cap}
//{POST: Calcula el tercer costat. Retorna l'àrea del triangle}
//{Calculada segons la fórmula de l'Herón}
double area(double& c) const;
```

Exemple: Classe Triangle, especificació (cont.)

```
    //{PRE: Cap}
    //{POST: Retorna l'àrea del triangle}
    //{Calculada segons la fórmula de b·h/2}
    double area() const;
    //Altres mètodes
};
#endif
```

Exemple: Classe Triangle, implementació

```
//Constructor
//{PRE: Rep els costats aa, bb i angle CC en radian}
//{POST: Construeix el triangle definit per aa, bb i CC}
Triangle::Triangle(double aa, double bb, double CC){
      a=aa;b=bb;C=CC;
}

//Constructor de còpia
//{PRE: Rep un objecte Triangle}
//{POST: Crea un objecte Triangle còpia de l'objecte donat}
Triangle::Triangle(const Triangle& T){
       a=T.a; b=T.b;C=T.C;
}
//Destructor per defecte
Triangle::~Triangle(){}

//Mètodes consultors
double Triangle::costat_a() const {return a;}
double Triangle::costat_b() const {return b;}
```

Exemple: Classe Triangle, implementació

```
double Triangle::costat_c() const{
  //teorema del cosinus
  return sqrt(a*a+b*b-2*a*b*cos(C));
}
double Triangle::angleA() const {
  //teorema del cosinus
  return asin(a*sin(C)/costat_c());
}
double Triangle::angleB() const{
  //teorema del cosinus
  return asin(b*sin(C)/costat_c());
}
double Triangle::angleC() const {return C;}

//{PRE: Rep els costats a, b i c del triangle}
//{POST: Retorna el perímetre del triangle}
double Triangle::perimetre() const {
   return a+b+costat_c();
   }
```

Exemple: Classe Triangle, implementació (cont.)

```
//{PRE: Cap}
//{POST: Calcula el tercer costat. Retorna l'àrea del triangle}
//{Calculada segons la fórmula de l'Herón}
double Triangle::area(double& c) const{
      //teorema de l'Herón
      c=costat_c();
      double p=perimetre()/2;
return  sqrt(p*(p-a)*(p-b)*(p-c));
}

//{PRE: Cap}
//{POST: Retorna l'àrea del triangle}
double Triangle::area() const{
   //teorema b·h/2
   double h= a*sin(C);
   return b*h/2;

}
```

Exemple: Classe Triangle, crida de mètodes sobrecarregats

```
int main(){
  Triangle T(3.0,4.0,Pi/2);

  cout << T.area() << endl;

  double c;
  cout << T.area(c) << endl;
  return 0;
}
```

Resum

- Els mètodes d'una classe són declarats com membres privats o públics de la mateixa
- Els mètodes públics formen l'API de la classe mentre que els mètodes privats són d'ús intern per a la classe
- L'especificació i la implementació d'un mètode de la classe es fa de la mateixa manera com la d'una funció de C++ en general
- Els mètodes tenen accés directe als altres membres de la classe
- Per tal d'evitar la modificació per error dels membres d'un objecte de la classe, els mètodes es declaren constants (amb la paraula clau const)
- Els mètodes es poden sobrecarregar (tenir diferents mètodes amb el mateix nom). S'ha de garantir que les llistes de paràmetres siguin diferents

Criteris pràctics. Evitar errors

- Es una bona pràctica especificar (encara que sigui en llenguatge natural) els mètodes
- Cal agrupar els mètodes en consultors i modificadors partint del criteri de consulta i modificació, respectivament
- Declarar els mètodes consultors **const**
- Fer privats aquells mètodes que serveixen per a tasques internes de la classe
- Usar la sobrecàrrega com a eina d'incrementar la llegibilitat del codi de la classe
- Assegurar que les llistes de paràmetres són diferents. Dos paràmetres d'una llista són diferents si difereixen en tipus no en nom.

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 6, McGraw-Hill, 2002.
- *Complementàries*:
  - Walter Savitch. *Resolución de problemas con C++*. Capítols 3 i 6, Prentice-Hall, 2000.

7. TADs: Implementació de TADs en C++. Operadors i sobrecàrrega

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - sobrecàrrega
  - operadors sobrecarregables (unaris, binaris)
  - operadors no sobrecarregables
  - operadors sobrecarregables com a membres
  - operadors sobrecarregables com a funció amiga
  - sobrecàrrega d'operador d'assignació
  - sobrecàrrega dels operadors del canal d'entrada i de sortida
  - exemple de sobrecàrrega: classe `string`
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- En les classes anteriors hem vist la sintaxi de la classe en C++
- Com a part de l'API del TAD hi ha també els operadors
- Veurem com fer la sobrecàrrega dels operadors en C++. Això ens permetrà usar els operadors habituals dels tipus bàsics per als tipus de dades compexes
- Els operadors que es poden sobrecarregar depenen de la classe concreta que s'està defininit; hi ha però, operadors com ara els d'entrada/sortida, el d'assignació i alguns operadors relacionals que es poden sobrecarregar per a qualsevol classe

Sobrecàrrega dels operadors

- Permet manipular els objectes amb els operadors habituals
- Es basa en la redefinició dels operadors predefinits del C++, sense modificar el seu sentit!

  En realitat els operadors estàndards ja estan sobrecarregats per als tipus predefinits
- Per exemple podem definir operadors per als nombres racionals, complexos, vectors, matrius, dates ...
- Es pot fer la sobrecàrrega dels operadors com mètodes o funcions
- Els operadors guarden l'ordre de prioritat establert.
- És necessari sobrecarregar alguns operadors per fer servir les estructures de dades i algorismes de l'STL

Operadores sobrecarregables

- La gran majoria (*excepte quatre!*) dels operadors són sobrecarregables
- Ens interessen:
  - operadors *aritmètics* (unari: $-$ i binaris: $+, -, *, /$)
  - operador d'*assignació* $(=)$
  - operadors *relacionals* $(<,\ <=,\ >,\ >=,\ ==,\ !=)$
  - operadors del canal *d'entrada/sortida* $(>>$ i $<<)$

Operadors no sobrecarregables

- accés a un membre **.**
- indirecció d'accés a un membre **.***
- resolució d'àmbit **::**
- operador ternari **?:**

Maneres de sobrecarregar els operadors

Hi ha dues maneres per sobrecarregar un operador:

- implementant-lo com a funció membre de la classe (la forma més usual)
- implementant-lo com a funció amiga (*friend*) de la classe

Hi ha certes diferències que cal tenir present.

Operadors sobrecarregables com a funcions membres

- Com a funció membre
  - l'operador es propi de la classe
  - té accés directe a les dades (com qualsevol altre membre)
  - ha de ser part de l'API de la classe (part pública)
- Sintaxi: `tipus operator op (llista paràmetres);`
  - `tipus` és el tipus de resultat, generalment del tipus classe que s'està definint, però potser un altre (ex. `int operator-(const Data& d1, const Data& d2);`).
  - `llista paràmetres` és buida en cas d'operadors unaris i d'un paràmetre en cas de binaris.

Exemple: operadors aritmètics per vectors 3D

```
//Especificació del TAD Vector3D. Fitxer Vector3D.hpp. VERSIÓ 1
#ifndef VECTOR3D_HPP
#define VECTOR3D_HPP
class Vector3D{
   double X,Y,Z; //coordenades del vector
  public:
   //Constructor per defecte. Crea vector V0(0.0,0.0,0.0)
   Vector3D();
   //Constructor amb paràmetres
   //{PRE: Rep tres reals XX, YY i ZZ}
   //{POST: Crea el vector amb coordenades XX, YY i ZZ}
   Vector3D(double XX, double YY, double ZZ);
   //Constructor de còpia. En aquest cas no fa falta!
   //{PRE: Rep un objecte V de tipus Vector3D}
   //{POST: Crea el vector còpia  de V}
   Vector3D(const Vector3D& V);
```

Exemple: operadors aritmètics per vectors 3D (cont.)

```
//Mètodes consultors

//Mètode per consultar la coordenada X del vector
//{PRE: Cap}
//{POST: Retorna la coordenada X del vector}
double getX() const;

//Mètode per consultar la coordenada Y del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Y del vector}
double getY() const;

//Mètode per consultar la coordenada Z del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Z del vector}
double getZ() const;

//Mètode per calcular el mòdul del vector
//{PRE: Cap}
//{POST: Retorna el mòdul del vector}
double modul() const;
```

Edicions UPC
Inici
Contingut
Pàgina 135
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: operadors aritmètics per vectors 3D (cont.)

```
//Mètodes modificadors
//Mètode per assignar l'abscisa del vector
//{PRE: Rep un real XX}
//{POST: Assigna a l'abscisa el valor XX}
void setX(double XX);
//Mètode per assignar l'ordenada del vector
//{PRE: Rep un real YY}
//{POST: Assigna a l'ordenada el valor YY}
void setY(double YY);
//Mètode per assignar la coordenada Z del vector
//{PRE: Rep un real ZZ}
//{POST: Assigna a la coordenada Z el valor ZZ}
void setZ(double ZZ);
//Altres mètodes
  //Mètodes per calcular projeccions del vector sobre els eixos / plans...
//Mètodes per calcular l'angle del vector amb els eixos / plans ...
//Operadors aritmètics
//Operadors unari canvi de signe
Vector3D operator-();
```

Exemple: operadors aritmètics per vectors 3D (cont.)

```
  //Suma de vectors
  Vector3D operator+(const Vector3D& V);
  //Resta de vectors
  Vector3D operator-(const Vector3D& V);
};
#endif
```

Edicions UPC
Inici
Contingut
Pàgina 136
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: operadors aritmètics per vectors 3D, implementació

```
//Implementació del TAD Vector3D. Fitxer Vector3D.cpp. VERSIÓ 1
#include<Vector3D.hpp>
#include<math.h>

  //Constructor per defecte. Crea vector V0(0.0,0.0,0.0)
  Vector3D::Vector3D():X(0.0),Y(0.0),Z(0.0)
  {}
  //Constructor amb paràmetres
  //{PRE: Rep tres reals XX, YY i ZZ}
  //{POST: Crea el vector amb coordenades XX, YY i ZZ}
  Vector3D::Vector3D(double XX, double YY, double ZZ){
    X=XX; Y=YY; Z=ZZ;
  }
  //Constructor de còpia
  //{PRE: Rep un objecte V de tipus Vector3D}
  //{POST: Crea el vector còpia  de V}
  Vector3D::Vector3D(const Vector3D& V){
   X=V.X; Y=V.Y;Z=V.Z;
  }
```

Exemple: operadors aritmètics per vectors 3D, implementació (cont.)

```
//Mètodes consultors

//Mètode per consultar la coordenada X del vector
//{PRE: Cap}
//{POST: Retorna la coordenada X del vector}
double Vector3D::getX() const {return X;}

//Mètode per consultar la coordenada Y del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Y del vector}
double Vector3D::getY() const {return Y;}

//Mètode per consultar la coordenada Z del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Z del vector}
double Vector3D::getZ() const {return Z;}
```

Exemple: operadors aritmètics per vectors 3D, implementació (cont.)

```
//Mètode per calcular el mòdul del vector
//{PRE: Cap}
//{POST: Retorna el mòdul del vector}
double Vector3D::modul() const{
   return sqrt(X*X+Y*Y+Z*Z);
}
//Mètodes modificadors

//Mètode per assignar l'abscisa del vector
//{PRE: Rep un real XX}
//{POST: Assigna a l'abscisa el valor XX}
void Vector3D::setX(double XX) {X=XX;}

//Mètode per assignar l'ordenada del vector
//{PRE: Rep un real YY}
//{POST: Assigna a l'ordenada el valor YY}
void Vector3D::setY(double YY) {Y=YY;}
```

Exemple: operadors aritmètics per vectors 3D, implementació (cont.)

```
//Mètode per assignar la coordenada Z del vector
//{PRE: Rep un real ZZ}
//{POST: Assigna a la coordenada Z el valor ZZ}
void Vector3D::setZ(double ZZ) {Z=ZZ;}
//Altres mètodes
//Mètodes per calcular projeccions del vector sobre els eixos/plans...
//Mètodes per calcular l'angle del vector amb els eixos / plans ...
//Operadors aritmètics
//Operadors unari canvi de signe
Vector3D Vector3D::operator-(){
  Vector3D V(-X,-Y,-Z);
  return V;
}
//Suma de vectors
Vector3D Vector3D::operator+(const Vector3D& V){
   Vector3D V1(X+V.X,Y+V.Y,Z+V.Z);
   return V1;
}
```

Exemple: operadors aritmètics per vectors 3D, implementació (cont.)

```
//Resta de vectors
Vector3D Vector3D::operator-(const Vector3D& V){
  Vector3D V1(X-V.X,Y-V.Y,Z-V.Z);
  return V1;
}
```

Edicions UPC
Inici
Contingut
Pàgina 141
Tornar
Pantalla Completa
Tancar
Sortir

Edicions UPC
Inici
Contingut
Pàgina 142
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: operadors aritmètics per vectors 3D, ús

```
int main(){
   Vector3D V0;
    cout << V0.getX() << V0.getY() << V0.getZ() << endl;
   V0.setX(1); V0.setY(1);

    V0=-V0;
    cout << V0.getX() << V0.getY() << V0.getZ() << endl;

    Vector3D i(1,0,0); Vector3D j(0,1,0);  Vector3D k(0,0,1);
    Vector3D V1=i+j; Vector3D V2=i+k; Vector3D V3=j+k;
    Vector3D V4=i+V3;
  return 0;
}
```

Funcions amigues

Les funcions amigues són funcions ordinàries amb dret d'accés *extraordinari* als membres privats d'una classe

- Una funció *amiga* no és una función membre de la classe però té accés als membres privats de la classe com si fos funció membre
- Motius?
  - tasca no pròpia de la classe
  - més eficiència
- Característiques:
  - s'especifiquen amb la paraula clau *friend* que es posa al principi del prototip de la funció
  - s'invoquen com qualsevol funció ordinària (no aplicables sobre l'objecte amb operador **.**)

Nota: La majoria dels operadors se solen declarar com funcions membre de la classe. Alguns (com el d'assignació =) és obligatori que siguin funcions membres

Edicions UPC
Inici
Contingut
Pàgina 144
Tornar
Pantalla Completa
Tancar
Sortir

Exemple declaració funció amiga

```
//Definició de la classe Data per a l'exemple de funció amiga
//per indicar si dues dates (p.ex. dos esdeveniments)
//cauen dins el mateix mes
class Data{
     int d,m,a;//dia, mes i any

   public:
      //Constructor per defecte
      Data();
      //Constructor amb paràmetres
      Data (int dd, int mm, int aa);
      //Destructor
      ~Data();

      //Mètode amiga
      friend bool DiesMateixMes(const Data& data1,
      const Data& data2);
      //Resta mètodes...
};
```

Exemple implementació funció amiga

```
// Constructor per defecte
Data::Data(){
   d=1;m=1;a=1601;
}
//Constructor amb paràmetres
Data::Data (int dd, int mm, int aa):d(dd),m(mm),a(aa){}

//Destructor
Data::~Data(){}

//Altres mètodes ...

//Mètode amiga
bool DiesMateixMes(const Data& data1, const Data& data2){
      return (data1.m==data2.m);//Accés directe als membres
}
```

Exemple d'ús funció amiga

```
int main(){

  Data D1(1,1,2003);
  Data D2(23,11,2005);

  if (DiesMateixMes(D1,D2)) cout << 'S'<< endl;
  else  cout << 'N';

 return 0;
}
```

Operadors sobrecarregables com a funcions amigues

- Com a funció amiga
  - encara que es declaren dins la classe, no són membres de la mateixa
  - té accés directe a les dades ja que és declarat funció amiga
  - es pot declarar en qualsevol part de la classe (privada, publica,...)
  - en la seva implementació no es relaciona amb l'àmbit de la classe

Nota: Els operadors que se solen declarar com funcions amigues són els operadors relacionals i els d'entrada/sortida.

Exemple: producte escalar i vectorial de vectors 3D

```
//Especificació del TAD Vector3D. Fitxer Vector3D.hpp. VERSIÓ 2
#ifndef VECTOR3D_HPP
#define VECTOR3D_HPP
class Vector3D{
   double X,Y,Z; //coordenades del vector
  public:
   //Constructor per defecte. Crea vector V0(0.0,0.0,0.0)
   Vector3D();
   //Constructor amb paràmetres
   //{PRE: Rep tres reals XX, YY i ZZ}
   //{POST: Crea el vector amb coordenades XX, YY i ZZ}
   Vector3D(double XX, double YY, double ZZ);
   //Constructor de còpia
   //{PRE: Rep un objecte V de tipus Vector3D}
   //{POST: Crea el vector còpia  de V}
   Vector3D(const Vector3D& V);
```

Exemple: producte escalar i vectorial de vectors 3D (cont.)

```
//Mètodes consultors
//Mètode per consultar la coordenada X del vector
//{PRE: Cap}
//{POST: Retorna la coordenada X del vector}
double getX() const;
//Mètode per consultar la coordenada Y del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Y del vector}
double getY() const;
//Mètode per consultar la coordenada Z del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Z del vector}
double getZ() const;
//Mètode per calcular el mòdul del vector
//{PRE: Cap}
//{POST: Retorna el mòdul del vector}
double modul() const;
```

Exemple: producte escalar i vectorial de vectors 3D (cont.)

```
//Mètodes modificadors
//Mètode per assignar l'abscisa del vector
//{PRE: Rep un real XX}
//{POST: Assigna a l'abscisa el valor XX}
void setX(double XX);
//Mètode per assignar l'ordenada del vector
//{PRE: Rep un real YY}
//{POST: Assigna a l'ordenada el valor YY}
void setY(double YY);
//Mètode per assignar la coordenada Z del vector
//{PRE: Rep un real ZZ}
//{POST: Assigna a la coordenada Z el valor ZZ}
void setZ(double ZZ);
//Altres mètodes
//Mètodes per calcular projeccions del vector sobre els eixos/plans
//Mètodes per calcular l'angle del vector amb els eixos/plans
```

Exemple: producte escalar i vectorial de vectors 3D (cont.)

```
  //Operadors aritmètics
  //Operadors unari canvi de signe
  Vector3D operator-();

  //Suma de vectors
  Vector3D operator+(const Vector3D& V);
  //Resta de vectors
  Vector3D operator-(const Vector3D& V);

  //Producte escalar de dos vectors
  friend double operator*(const Vector3D& V1,const Vector3D& V2);

  //Producte vectorial de dos vectors
  friend Vector3D producteV(const Vector3D& V1,const Vector3D&
  V2);
};
#endif
```

Exemple: producte escalar i vectorial de vectors 3D, implementació

```
//Implementació del TAD Vector3D. Fitxer Vector3D.cpp. VERSIÓ 2
#include<...>

  //Constructor per defecte. Crea vector V0(0.0,0.0,0.0)
  Vector3D::Vector3D():X(0.0),Y(0.0),Z(0.0)
  {
  }
  //Constructor amb paràmetres
  //{PRE: Rep tres reals XX, YY i ZZ}
  //{POST: Crea el vector amb coordenades XX, YY i ZZ}
  Vector3D::Vector3D(double XX, double YY, double ZZ){
     X=XX; Y=YY; Z=ZZ;
  }
  //Constructor de còpia
  //{PRE: Rep un objecte V de tipus Vector3D}
  //{POST: Crea el vector còpia  de V}
  Vector3D::Vector3D(const Vector3D& V){
    X=V.X; Y=V.Y;Z=V.Z;
```

Exemple: producte escalar i vectorial de vectors 3D, implementació (cont.)

```
//Mètodes consultors

//Mètode per consultar la coordenada X del vector
//{PRE: Cap}
//{POST: Retorna la coordenada X del vector}
double Vector3D::getX() const {return X;}

//Mètode per consultar la coordenada Y del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Y del vector}
double Vector3D::getY() const {return Y;}

//Mètode per consultar la coordenada Z del vector
//{PRE: Cap}
//{POST: Retorna la coordenada Z del vector}
double Vector3D::getZ() const {return Z;}
```

Exemple: producte escalar i vectorial de vectors 3D, implementació (cont.)

```
//Mètode per calcular el mòdul del vector
//{PRE: Cap}
//{POST: Retorna el mòdul del vector}
double Vector3D::modul() const{
   return sqrt(X*X+Y*Y+Z*Z);
}
//Mètodes modificadors

//Mètode per assignar l'abscisa del vector
//{PRE: Rep un real XX}
//{POST: Assigna a l'abscisa el valor XX}
void Vector3D::setX(double XX) {X=XX;}

//Mètode per assignar l'ordenada del vector
//{PRE: Rep un real YY}
//{POST: Assigna a l'ordenada el valor YY}
void Vector3D::setY(double YY) {Y=YY;}
```

Edicions UPC
Inici
Contingut
Pàgina 155
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: producte escalar i vectorial de vectors 3D, implementació (cont.)

```
//Mètode per assignar la coordenada Z del vector
//{PRE: Rep un real ZZ}
//{POST: Assigna a la coordenada Z el valor ZZ}
void Vector3D::setZ(double ZZ) {Z=ZZ;}

//Altres mètodes
//Mètodes per calcular projeccions del vector sobre els eixos/plans
//Mètodes per calcular l'angle del vector amb els eixos /plans
//Operadors aritmètics

//Operador unari canvi de signe
Vector3D Vector3D::operator-(){
  Vector3D V(-X,-Y,-Z);
  return V;
}
//Suma de vectors
Vector3D Vector3D::operator+(const Vector3D& V){
   Vector3D V1(X+V.X,Y+V.Y,Z+V.Z);
   return V1;
}
```

Exemple: producte escalar i vectorial de vectors 3D, implementació (cont.)

```
//Resta de vectors
Vector3D Vector3D::operator-(const Vector3D& V){
  Vector3D V1(X-V.X,Y-V.Y,Z-V.Z);
  return V1;
}
```

```
//Producte escalar de dos vectors
double operator*(const Vector3D& V1, const Vector3D& V2){
  return (V1.X*V2.X+V1.Y*V2.Y+V1.Z*V2.Z);
}
```

```
//Producte vectorial de dos vectors
Vector3D producteV(const Vector3D& V1,const Vector3D& V2){
      Vector3D  V(V1.Y*V2.Z-V1.Z*V2.Y,  V1.Z*V2.X  -  V1.X*V2.Z,
V1.X*V2.Y-V1.Y*V1.X);
  return V;}
```

Exemple: producte escalar i vectorial de vectors 3D, ús

```
int main(){
   Vector3D V0;
    cout << V0.getX() << V0.getY() << V0.getZ() << endl;
   V0.setX(1); V0.setY(1);
    V0=-V0;
    cout << V0.getX() << V0.getY() << V0.getZ() << endl;

    Vector3D i(1,0,0); Vector3D j(0,1,0);  Vector3D k(0,0,1);
    Vector3D V1=i+j; Vector3D V2=i+k; Vector3D V3=j+k;
    Vector3D V4=i+V3;
    cout << j*k << endl;

    Vector3D V5=producteV(i,j);
  return 0;
}
```

Sobrecàrrega d'operador d'assignació

- Per defecte, C++ sobrecarrega l'operador d'assignació = per a classes definides per l'usuari
- Per exemple, V2=V1; per a vectors, copia membre a membre les dades de V1 a membres de V2
- L'assignació d'un objecte d'una classe a un altre de la mateixa classe es realitza fent servir l'*operador d'assignació de còpia* (és essencialment el mateix que la inicialització per defecte membre a membre, però no fa servir el constructor de còpia)
- Cal anar amb compte per no fer una auto-assignació!
- Sintaxi:

```
nomClasse& nomClasse::operator=(const nomClasse& Obj){
   //test per evitar l'auto-assignació
   if(this!=&Obj){
      //realitzem la còpia
   }
   //retornem l'objecte resultat d'assignació
   return *this;
}
```

Observacions

- En implementar aquest operador, podem necessitar *alliberar recursos* i *obtenir novament recursos* (ho veurem pel cas de membres taules)
- Cal copiar tots els continguts
- Retornar-se a si mateix
- Ha de ser funció membre!

Exemple: operador d'assignació per vectors 3D

- *En la declaració de la classe:*

```
//Operador d'assignació. No és necessari en aquest cas
Vector3D& operator=(const Vector3D& V);
```

- *En la implementació de la classe:*

```
//Operador d'assignació. No és necessari en aquest cas
Vector3D& Vector3D::operator=(const Vector3D& V){
  if(this!=&V){
    X=V.X;
    Y=V.Y;
    Z=V.Z;
   }
  return *this;
}
```

- *L'ús de l'operador sobrecarregat:* el de sempre!

Sobrecàrrega dels operadors relacionals

- Els operadors relacionals se sobrecarreguen de manera similar ja sigui com funcions membre o funcions amigues
- En general és bo implementar el == (del qual podem derivar el != o l'implementem igualment)
- Si la classe definida ho permet podem sobrecarregar els altres operadors (com ara el <, <=). De fet, si volem usar algorismes de biblioteca (ex. d'ordenació) cal proporcionar-los
- Generalment se sobrecarreguen com funcions amigues ja que són operacions externes al tipus

Exemple: operadors relacionals per vectors 3D

- *En la declaració de la classe:*

```
//Operadors relacionals
  friend bool operator==(const Vector3D& V1,const Vector3D&
V2);
 friend bool operator!=(const Vector3D& V1, const Vector3D& V2);
```

- *En la implementació de la classe:*

```
//Operadors relacionals
bool operator==(const Vector3D& V1,const Vector3D& V2){
     return (V1.X==V2.X && V1.Y==V2.Y && V1.Z==V2.Z);
}
bool operator!=(const Vector3D& V1,const Vector3D& V2){
     return (V1.X!=V2.X ||V1.Y!=V2.Y ||V1.Z==V2.Z);
}
```

- *L'ús de l'operador sobrecarregat:* el de sempre!

```
if (-V0==V1) cout << 'S'; else cout << 'N';
```

Sobrecàrrega dels operadors del canal d'entrada / sortida

- Es poden sobrecarregar els operadors $<<$ (d'inserció) i $>>$ (d'extracció) de la `iostream`
- Es pot manipular qualsevol sentència de fluxe de qualsevol objecte de classe
- Faciliten molt la lectura i escriptura dels objectes amb molta informació així com el formateig de la sortida
- Se sobrecarreguen com a funcions amigues (ja que no corresponen a una tasca/responsabilitat pròpiament dita de la classe)

Exemple: operadors del canal d'entrada/sortida per vectors 3D

- *En la declaració de la classe:*

```
//Operador del canal d'entrada
friend istream& operator>>(istream& is,Vector3D& V);
//Operador del canal de sortida
friend ostream& operator<<(ostream& os,const Vector3D& V);
```

- *En la implementació de la classe:*

```
//Operador del canal d'entrada
istream& operator>>(istream& is,Vector3D& V){
        is>>V.X>>V.Y>>V.Z;
        return is;
}
//Operador del canal de sortida
ostream& operator<<(ostream& os,const Vector3D& V){
     os<<V.X<<' '<<V.Y<<' '<<V.Z;
     return os;}
```

- *L'ús de l'operador sobrecarregat:* el de sempre!

```
Vector3D V7;
cin>>V7;
cout << V7;
```

Exemple de sobrecàrrega: classe `string` de la biblioteca estàndard

- `string` és un tipus definit en la lliberia estàdard STL per a manipular cadenes de caràcters
- Com a classe, string té sobrecarregats molts operadors que faciliten l'ús dels objectes del tipus. Així tenim:
  - operador d'assignació que ens permet fer:
    ```
    string s1="Hola"; string s2=s1;
    ```
  - operador + que ens permet concatenar strings:
    ```
    string s3="que"; sring s4="tal!"; string s5=s1+s3+s4;
    ```
  - operador d'accés directe []:
    ```
    cout << "Lletra a la posicio 3 del s1:  " << s1[2];
    ```
  - operadors de comparació ==, != i els d'ordre alfabètic (< <= > >=)
  - operadors d'entrada/sortida que permeten llegir pel teclat i escriure a la pantalla cadenes de caràcters

Resum

- El llenguatge C++ permet sobrecarregar els operadors usuals del llenguatge per tal de treballar amb els objectes de tipus no bàsics
- La sobrecàrrega la podem fer o bé amb operadors membres o bé amb mètodes amigues
- Els operadors de la classe faciliten la lectura dels programes a més de produir programes més compactes
- Es operadors que hem d'implementar depenen de la classe. Així, per a alguna classe com la de Vector3D té sentit definir operadors aritmètics
- En tot cas, els operadors d'assignació, comparació i entrada/sortida es poden definir per qualsevol classe

Criteris pràctics. Evitar errors

- No es pot modificar el significat dels operadors predefinits (respecte al nombre d'operands, i prioritat)
- Alguns dels paràmetres d'un operador sobrecarregat ha de ser d'un tipus classe (per evitar canviar el significat de l'operador)
- No es poden crear nous operadors!
- Un criteri que se sol aplicar per triar la sobrecàrrega per funció membre o funció amiga és pel tipus de resultat de l'operador:
  - els que retornen un objecte de la mateixa classe (operacions internes) s'implementen com funcions membres i,
  - els que no (operacions externes) s'implementen com funcions amigues

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 15, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).
  - Walter Savitch. *Resolución de problemas con C++.* Capítol 8, Prentice-Hall, 2000.

8. Apuntadors i gestió de memòria dinàmica

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - Adreces de memòria i referències (repàs)
  - Apuntadors. Declaració dels apuntadors
  - Indirecció dels apuntadors
  - Inicialització dels apuntadors
  - Apuntadors i verificació de tipus
  - Aritmètica d'apuntadors
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 170
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- Tot programa opera amb objectes
- Els objectes que sabem declarar i definir fins ara són objectes en memòria estàtica
- El programa i les variables globals resideixen en memòria estàtica. També hi ha la pila d'execució de programa
- La memòria estàtica té limitacions, la reserva de la memòria es fa en temps de compilació
- Cal una manera de reservar / alliberar memòria per les variables en *temps d'execució* del programa. Això és el *Heap* o *Free Store*
- Cal un nou tipus de dades: el tipus punter, els valors del qual són adreces de memòria

Adreces de memòria i referències (repàs)

- Cada objecte d'un programa es troba en una zona de la memòria de la qual sabem l'*adreça* (del primer byte).
- Quan es *declara* una variable, el compilador hi associa:
  - el nom de la variable, a través del qual li fem referència
  - una adreça de memòria. L'adreça es després usada per accedir al valor de la variable
- Per exemple:

`int n;` {
- `0x241ff5c`, adreça de memòria (direcció del byte inicial)
- `n`, identificador de la variable
- `int`, tipus de dades de `n`
- informa el compilador sobre el tamany

- Quan es defineix la variable (declaració i definició són diferents en C++), en l'espai de la memòria s'emmagatzema el valor

Exemple

L'adreça a la memòria d'un objecte es retorna per l'operador & (per exemple &n retorna 0x241ff5c)

```
int main(){
    int n;
    cout << "Adreca de n: "<< &n << endl;
    cout << "sizeof(n): "<< sizeof(n) << endl;
    n=12;
    cout << "Valor de n: "<< n << endl;
   int x[10];
    cout << "Adreca de x: "<< x << "= "<< &x[0]
    << "Adreca de x[0]"<< endl;
    cout << "sizeof(x): "<< sizeof(x) << endl;
    cout << "Adreca de x[5]: "<< &x[5]
    << "= "<< x+5 << "Adreca de x+5"<< endl;
    return 0;
}
```

Sortida del programa

```
Adreca de n:  0x241ff5c
sizeof(n):  4

Valor de n:  12

Adreca de x:  0x241ff20 = 0x241ff20 Adreca de x[0]
sizeof(x):  40
Adreca de x[5]:  0x241ff34 = 0x241ff34 Adreca de x+5
```

Operador d'adreça. Referències (repàs)

- L'operador d'adreça té una altra utilitat: permet definir variables de *tipus referència*
- Una referència a una variable és un *"alias"* (sinònim o un nom alternatiu) a la variable a la qual es refereix
- Una referència s'ha d'inicialitzar sempre!

```
int u=23;
int& ru=u;

cout << "u=" << u << "ru =" << ru << endl;
cout << "&u=" << &u << "&ru =" << &ru << endl;

ru =23 &u=0x241ff1c
&ru=0x241ff1c
```

Edicions UPC
Inici
Contingut
Pàgina 176
Tornar
Pantalla Completa
Tancar
Sortir

Apuntadors: concepte

- L'adreça d'un objecte a la memòria es pot guardar a una variable, aquest tipus de variable s'anomena *variable de tipus apuntador o punter*
- Una variable apuntador (*punter*) és una variable que conté adreces de memòria d'altres variables. O sigui, el tipus de valor que manté un apuntador és adreces de memòria on s'emmagatzemen dades
- El significat comú és que un apuntador *"apunta a"* la posició de memòria d'un altre objecte
- Utilitat dels apuntadors: accés i manipulació eficient de dades

Apuntadors: declaració

- Sintaxi de declaració d'una variable punter:

```
T* pT;
```

  `pT` –variable que emmagatzema l'adreça a la memòria d'objectes de tipus `T`

- Remarquem: tenim, bàsicament, un tipus punter per cada tipus de dades (punter a enter, punter a caràcter, punter a Pixel, etc.)
- També hi ha un tipus punter `void` (no associat a un tipus de dades concret) que no veurem com tampoc veurem punters a punters

Exemple declaració apuntadors

```
    int n = 10;
    int* pn = &n; // punter a n, emmagatzema l'adreça de n

    cout << "n = "<< n << endl;
    cout << "Adreca de n: "<< &n <<endl;
    cout << "Valor emmagatzemat en el punter pn: "<< pn ;
    cout << "Adreca del punter pn mateix: "<< &pn ;
     return 0;
}
```

dóna:

```
n = 10
Adreca de n:  0x241ff5c
Valor emmagatzemat en el punter pn:  0x241ff5c
Adreca del punter pn mateix:  0x241ff58
```

Edicions UPC
Inici
Contingut
Pàgina 179
Tornar
Pantalla Completa
Tancar
Sortir

Apuntadors: desreferenciació (indirecció)

- Si tenim que `T* pT;` llavors `pT` apunta a un objecte de tipus `T`. Com podem "agafar" l'objecte mateix al qual apunta `pT`?
- L'operador de *desrefèrencia* `*` ens permet obtenir l'objecte
- Sintaxi: cal utilitzar en una expressió `*pT`

```
*pn=15;//escriu el valor 15 a l'adreça de memòria emmagatzemat en pn
cout << "El valor de n= "<< n << endl; // n té valor 15!
```

- Un punter pot apuntar cap objecte, és un punter nul (de valor 0)

  Nota: Intentar desreferenciar un punter amb valor NULL és un error greu!

Edicions UPC
Inici
Contingut
Pàgina 180
Tornar
Pantalla Completa
Tancar
Sortir

- Alternativament, quan l'objecte apuntat per pT és objecte d'una classe, podem utilitzar l'operador $\rightarrow$ per accedir als membres públics de la classe.

Un altre exemple

```
int main() {
  double x = 21.5;
  double* px; // punter a una variable real
  px = &x;    // a px li assignem l'adreça a la memòria de x
              // px apunta a x

  cout << "El valor de la variable x = "<< x << endl;
  cout << "El valor de la variable px = "<< px << endl;
  cout << "El valor apuntat per "<< px << "es "<< *px << endl;
  return 0;
}
```

```
El valor de la variable x = 21.5
El valor de la variable px = 0x241ff58
El valor apuntat per 0x241ff58 es 21.5
```

Exemple

Considerem ara el cas de punters a objectes d'una classe. Vegem primer la definició de la classe Pixel.

```
class Pixel{
      double R, G, B;
  public:
    Pixel(){}
    Pixel(double RR,double GG,double BB){
          R=RR; G=GG; B=BB; }
    Pixel(const Pixel& P){
     R = P.R; G = P.G; B = P.B; }
    double getRed(){return R;}
    double getGreen(){return G;}
    double getBlue(){return B;}

    void setRed(double RR){R=RR;}
    void setGreen(double GG){G=GG;}
    void setBlue(double BB){B=BB;}
  };
```

Edicions UPC
Inici
Contingut
Pàgina 182
Tornar
Pantalla Completa
Tancar
Sortir

Exemple (cont.)

```
int main() {
Pixel Pix(0.6,0.1,0.3); //es crea un objecte Pix de la classe Pixel
Pixel* pPix = &Pix; // pPix apunta a Pix

cout << "Els valors de pixel son: ";
cout << " "<< Pix.getRed(); cout << " "<< Pix.getGreen();
cout << " "<< Pix.getBlue() << endl;

cout << "Pix esta apuntat per "<< pPix << endl;

cout << "Els valors de pixel, un altre cop, son: ";
cout <<" "<<pPix->getRed()<<" "<<pPix->getGreen();
cout <<" "<<pPix->getBlue()<< endl;

  return 0;
  };
```

Sortida del programa

```
Els valors de pixel son:  0.6 0.1 0.3
Pix esta apuntat per 0x22ff48
Els valors de pixel, un altre cop, son:  0.6 0.1 0.3
```

Inicialització dels apuntadors

- Un cop declarat un apuntador, el següent pas és inicialitzar-lo i és llavors quan es pot usar per accedir a l'objecte al qual apunta.
- Un apuntador pot ser inicialitzat amb l'adreça de memòria d'un objecte (que ha ser del mateix tipus al que apunta).
- Potser l'adreça d'una variable individual o la del primer element d'una taula.
- Per inicialitzar un apuntador fem servir l'operador &:

```
T v; //declara la variable v de tipus T
T* pT; // declara un punter de tipus T
pT=&v; //assigna l'adreça de v a pT, pT apunta a v
```

Inicialització de punters

Punter inicialitzat a partir de:	Declaració i inicialització	Declaració i inicialització desdoblada
Un objecte individual `T x;`	`T* px = &x;`	`T* px; px=&x;`
Un array d'objectes `T x[15];`	`T* px = &x[0];`	`T* px; px=&x[0];`
Un altre punter del mateix tipus (ja inicialitzat) `T* py;`	`T* px = py;`	`T* px; px=py;`
Punter nul	`T* px = 0;`	`T* px; px=0;`
Un literal	`T* px = [literal];`	`T* px; px=[literal];`

Notes:

- Un punter nul apunta a cap objecte (per ser precisos apunta a una zona de memòria especial)
- La declaració d'un punter no implica reservar memòria (excepte per emmagatzemar una adreça)!
- Hi ha un segon objecte que aporta l'adreça
- Falta veure la inicialització a través de memòria dinàmica!

Exemple d'assignació de punters

```
int n = 10;
  int* pn = &n; // punter a n, emmagatzema l'adreça de n
  int* pn1 = pn; // On apunta pn1?

  cout << "Valor apuntat per pn: "<< *pn << endl;
  cout << "Valor apuntat per pn1: "<< *pn1 << endl;
  cout << " pn emmagatzema l'adreca: "<< pn << endl;
  cout << " pn1 emmagatzema l'adreca: "<< pn1 << endl;

  *pn1 = 20; // a quin valor apunten pn i pn1?

  cout << "Valor apuntat per pn: "<< *pn << endl;
  cout << "Valor apuntat per pn1: "<< *pn1 << endl;
  cout << "pn guarda l'adreca: "<< pn << endl;
  cout << "pn1 guarda l'adreca: "<< pn1 << endl;
```

Sortida del programa:

```
Valor apuntat per pn:  10
Valor apuntat per pn1:  10
pn emmagatzema l'adreca:  0x241ff5c
pn1 emmagatzema l'adreca:  0x241ff5c
Valor apuntat per pn:  20
Valor apuntat per pn1:  20
pn guarda l'adreca:  0x241ff5c
pn1 guarda l'adreca:  0x241ff5c
```

Què fa el següent programa?

```
#include<...>

int main() {
  char c;
  char* pc = &c;
   for (c='a';c<='z';c++)
       cout << *pc << "\t";

  return 0;
}
```

Apuntadors i verificació de tipus

- Els punters s'enllacen a tipus de dades específics. C++ fa una comprovació estricte!
- A un punter a `double` no se li pot assignar l'adreça d'un enter

```
double* pd;
int x=10;
pd=&x;//Error!
```

- Ara, el valor d'una variable punter, es pot canviar fent així que el punter apunti a diferents objectes del mateix tipus

Edicions UPC
Inici
Contingut
Pàgina 189
Tornar
Pantalla Completa
Tancar
Sortir

Arirtmètica d'apuntadors

- El tipus punter té associades algunes operacions aritmètiques
- Com que els valors de tipus són adreces de memòria, només són operacions legals les que donen adreces vàlides
  - es pot sumar o restar una *constant* punter a un punter
  - es pot sumar o restar un enter
- No es poden sumar, restar, multiplicar ni dividir els punters entre ells
- Aplicable als arrays

Edicions UPC
Inici
Contingut
Pàgina 190
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: convertir les lletres minúscules d'una cadena a majúscules

```
void passarAMaj(char text[]){
   char*p; //punter a caràcter per recórrer el text
   //Apuntem al primer caràcter del text
    p=&text[0];

    //Recorregut fins trobar-se caràcter NULL
    while (*p){
      if('a'<=*p &&*p<='z')
         *p=*p-32; //passem la lletra apuntada per p a majúscules
      p++; //següent caràcter
    }
}
```

Exemple: programa de prova

```
int main(){
    char text[80]; //Cadena del text

    cout << "Introdueix una cadena de caràcters a convertir:";
    cin.getline(text,sizeof(text));
    //crida de l'acció
    passarAMaj(text);
    cout << text << endl;
     return 0;
}
```

Resum

- Un tipus punter és un tipus de dades els valors del qual són adreces de memòria on resideixen els objectes del programa
- Un objecte punter ha de ser d'un tipus especificat: és capaç d'apuntar només als objectes d'un tipus
- El punter proporciona un mecanisme per accedir als objectes
- Cal tenir present la sintaxi de declaració i els operadors de desreferència i de referència
- Un punter es pot inicialitzar de diverses maneres, aquí hem vist la manera en què l'adreça que emmagatzema el punter és la d'un altre objecte
- Un punter de valor zero apunta a cap objecte, s'anomena punter nul

Criteris pràctics. Evitar errors

- Per accedir al contingut de l'objecte apuntat pel punter, cal que el punter sigui no nul.
- La sentència:

  `if (p)//...`

  comprovaria si `p` es diferent de nul, cas en què podem accedir al contingut de l'objecte apuntat per `p`
- En el cas d'un objecte, un punter per apuntar a l'objecte s'ha d'inicialitzar a l'adreça de l'objecte. En cas d'una taula s'ha d'inicialitzar a l'adreça del primer element de la taula
- En el cas de taules podem recorrer la aula usant un punter donada la contigüitat de les cel·les de la mateixa

Edicions UPC
Inici
Contingut
Pàgina 193
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 9, McGraw-Hill, 2002.
- *Complementàries*:
  - Walter Savitch. *Resolución de problemas con C++*. Capítol 11, Prentice-Hall, 2000.

9. TADs: Apuntadors i gestió de memòria dinàmica

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - operadors de la memòria dinàmica **new** i **delete**
  - ús de new i delete en constructors, destructor; en constructor de còpia i operador d'assignació
  - ús de new i delete en mètodes modificadors
- **Resum**
- **Referències**

Motivació. Relació amb el tema anterior

- En la classe anterior hem vist com declarar punters d'un tipus donat
- Declarar un punter no significa reservar memòria (tret per emmagatzemar una adreça
- La gestió de la memòria dinàmica (reservar / alliberar memòria) en C++ es fa amb els operadors **new** i **delete** del llenguatge
- L'ús de la memòria dinàmica permet implementar programes eficients i també comporta certa complexitat a l'hora de programar i depurar programes

Operadors de la memòria dinàmica *new* i *delete*

- A vegades no sabem abans d'executar el programa quanta memòria reservar. Amb l'assignació estàtica cal preveure la quantitat de memòria necessària!
- Així, per exemple, en cas de les taules parcialment omplertes, malgastem espai

  En lloc de fer `double notes[61];` que reserva l'espai de memòria en temps de compilació, voldríem fer (però no podem!) demanar a l'usuari introduir el tamany de la taula amb `cin » NEstudiants;` i reservar espai per `NEstudiants` reals en temps d'execució
- Solució: assignació dinàmica de memòria. C++ ofereix els operadors **new** i **delete** per gestionar la memòria dinàmica
- Amb aquests operadors s'obté (reserva) memòria – *en temps d'execució*– en crear objectes i s'allibera memòria destruint-los

Edicions UPC
Inici
Contingut
Pàgina 198
Tornar
Pantalla Completa
Tancar
Sortir

Operadors de la memòria dinàmica *new* i *delete* (cont.)

- L'operador **new** permet crear objectes d'una classe en memòria dinàmica
- Per crear un taula d'objectes, cal utilitzar el **new**[ ]
- L'operador **delete** permet destruir objectes en memòria dinàmica alliberant així memòria
- Per destruir una taula d'objectes, cal utilitzar el **delete**[ ]

Operadors de la memòria dinàmica: com funcionen?

new reserva l'espai en memòria i invoca el constructor de la classe per crear/inicialitzar l'objecte creat

- Sintaxi:

```
// per als tipus bàsics, estructures i classes
T* pT = new T(valor inicial opcional);
```

- L'operador new retorna un punter que conté l'adreça del bloc assignat de memòria (què passa si no s'aconsegueix fer l'assignació?)

new[ ] funciona similarment, s'invoca el constructor per defecte sobre cadascun dels objectes de la taula (remarquem la importància que la classe tingui el constructor per defecte!)

Operadors de la memòria dinàmica: com funcionen? (cont.)

- Sintaxi:

```
T* pT1 = new T[tamany];// serveix per als arrays
```

- La inicialització (valor inicial opcional) no es pot fer pels arrays

delete aplica el destructor de la classe i allibera la memòria

- Sintaxi: `delete pT;`

delete[ ] aplica el destructor de la classe per cadascun dels objectes de la taula i allibera la memòria.

- Sintaxi: `delete [] pT1;`

Observació: **new** i **delete** van en parella

Edicions UPC
Inici
Contingut
Pàgina 201
Tornar
Pantalla Completa
Tancar
Sortir

Exemple d'ús (tipus bàsics)

```
#include<...>
int main() {
  int* p = new int; //S'obté espai per a un enter
  cout << "Valor apuntat per p? "<< *p << endl; //quin és el valor?

  int* p1 = new int[10]; //S'obté espai per a una taula de 10 enters
  cout << "Valors a la taula?"<< endl;
  for (int i=0; i<10;i++){
      p1[i]=i;
      cout << p1[i] << "\t";
  }
  delete p;
  delete[] p1;
return 0;
}
```

Edicions UPC
Inici
Contingut
Pàgina 202
Tornar
Pantalla Completa
Tancar
Sortir

Exemple d'ús (classes)

```
int main() {
  //S'obté espai per a un nou objecte de la classe Estudiant
  Estudiant* E = new Estudiant("Pep", "Garcia", 1984, "So i Imatge");
  cout <<"Estudiant? "<<(*E).getNom()<<' '<<(*E).getCognom();
  //Alternativament, podem utilitzar l'operador → per accedir
  // als membres públics de la classe
   //s'obté espai per a una taula de 10 estudiants
  Estudiant *TE = new Estudiant[10];
  //...
  Estudiant E1 = TE[0];// accedim al primer estudiant
  cout<<"Data naixement primer estudiant? "
  <<E1.getDataNaix()<<endl;
  delete E;
  delete E1;// Error! No és un objecte en memòria dinàmica
  delete[] TE;
return 0;
}
```

Ús de new i delete en constructors i destructor

- Una classe pot tenir atributs que s'han d'assignar espai de memòria amb `new` o `new[]` i s'han de destruir amb el `delete` o `delete[]`, respectivament.
- **new** l'utilitzem en el constructors i **delete** l'utlitzem en el destructor
- Per exemple, en la classe `Estudiant` voldrem mantenir informació del seu nom, cognom, titulació i les seves notes, sense saber *apriori* la llargària de nom i cognom i quantes notes tindrà
- Podem optar a crear el nom, cognom, titulació i taula de notes en memòria dinàmica

Exemple: Especificació classe Estudiant

```
//Especificació del TAD Estudiant. Fitxer Estudiant.hpp
#ifndef ESTUDIANT_HPP
#define ESTUDIANT_HPP
#include<iostream>
class Estudiant{

    long int NIA; //número identificació alumne
    char* nom; //nom de l'estudiant
    char* cognom; //cognom de l'estudiant
    long int dataNaix; //data naixement de l'estudiant
    char* titulacio; ///nom titulació de l'estudiant
    double* notes; //taula de les notes de l'estudiant
    int NAssig;    // nombre d'assignatures de l'estudiant

 public:
    //Constructor per defecte. Crea un objecte tipus Estudiant
    //amb NIA=0 i els camps cadenes de caràcters buits
    Estudiant();
```

Exemple: Especificació classe Estudiant (cont.)

```
//Constructor amb paràmetres. Crea un objecte tipus Estudiant
// amb les dades proporcionades. Reserva memòria pels membres
//cognom, titulacio i notes
Estudiant(long int Id,char* N,char* C, long int data,
                    char* T,int NAss);
//Destructor. Allibera memòria adquirida
~Estudiant();

//Mètodes consultors

//Mètode per consultar el NIA
//{PRE: Cap}
//{POST: Retorna el NIA de l'estudiant}
long int getNIA();

//Mètode per consultar el nom
//{PRE: Cap}
//{POST: Retorna el nom de l'estudiant}
char* getNom();
```

Exemple: Especificació classe Estudiant (cont.)

```
//Mètode per consultar el cognom
//{PRE: Cap}
//{POST: Retorna el cognom de l'estudiant}
char* getCognom();

//Mètode per consultar la data de naixement
//{PRE: Cap}
//{POST: Retorna la data de naixement de l'estudiant}
long int getDataNaix();
//Mètode per consultar la titulació
//{PRE: Cap}
//{POST: Retorna la titulació de l'estudiant}
char* getTitulacio();
//Mètode per consultar la taula de notes
//{PRE: Cap}
//{POST: Retorna la taula de notes de l'estudiant}
double* getNotes();
//Mètode per consultar la nota i-èssima
//{PRE: Cap}
//{POST: Retorna la nota i-èssima de l'estudiant}
double Estudiant::getNota_i(int i);
```

Exemple: Especificació classe Estudiant (cont.)

```
//Mètodes modificadores
//Mètode per modificar el NIA
//{PRE: un enter llarg Id}
//{POST: Assigna Id al NIA de l'estudiant}
long int setNIA(long int Id);
//Mètode per modificar el nom
//{PRE: una cadena de caràcters N}
//{POST: Assigna N al nom de l'estudiant}
void setNom( char* N);
//Mètode per modificar el cognom
//{PRE: una cadena de caràcters C}
//{POST: Assigna C al cognom de l'estudiant}
void setCognom(char* C);

//Mètode per modificar la data de naixement
//{PRE: Rep un enter llarg data}
//{POST: Assigna la data a la data de naixement de l'estudiant}
void setDataNaix(long int data);
```

Exemple: Especificació classe Estudiant (cont.)

```
        //Mètode per modificar la titulació
        //{PRE: una cadena de caràcters T}
        //{POST: Assigna la T a la titulació de l'estudiant}
        void setTitulacio(char* T);
        //Mètode per modificar la taula de notes
        //{PRE: Un valor real nota i l'índex i de la nota a modificar}
        //{POST: Assigna  nota a notes[i] de notes de l'estudiant}
        void setNota_i(double nota,int i);

        //Operador de sortida
        friend ostream& operator<<(ostream& os,const Estudiant& E);
};
#endif
```

Edicions UPC
Inici
Contingut
Pàgina 208
Tornar
Pantalla Completa
Tancar
Sortir

Exemple: Implementació classe Estudiant

```
//Implementació del TAD Estudiant. Fitxer Estudiant.cpp
#include<Estudiant.hpp>

    //Constructor per defecte. Crea un objecte tipus Estudiant
    //amb NIA=0 i els camps cadenes de caràcters buits
    Estudiant::Estudiant():NIA(0),nom(""),cognom(""),titulacio(""){}

    //Constructor amb paràmetres.Crea un objecte tipus Estudiant
    // amb les dades proporcionades. Reserva memòria
    //pels membres nom, cognom, titulació i notes
    Estudiant::Estudiant(long int Id,char* N,char* C, long int data,
                         char* T,int NAss){
       NIA=Id;
       //Adquirim memòria pel nom
       nom=new char[strlen(N)+1];
       //Copiem continguts
       strcpy(nom,N);
```

Exemple: Implementació classe Estudiant (cont.)

```
    //Adquirim memòria pel cognom
    cognom=new char[strlen(C)+1];
    //Copiem continguts
    strcpy(cognom,C);
    dataNaix=data;
    //Adquirim memòria per la titulació
    titulacio=new char[strlen(T)+1];
    //Copiem continguts
    strcpy(titulacio,T);

    NAssig=NAss;
    //Adquirim memòria per la taula de notes
    notes=new double[NAssig];
    //Llegim notes pel canal d'entrada
    for(int i=0;i<NAssig;i++) cin>>notes[i];
}
```

Exemmple: Implementació classe Estudiant (cont.)

```
//Destructor. Allibera memòria adquirida
Estudiant::~Estudiant(){
   delete[] nom;
   delete[] cognom;
   delete[] titulacio;
   delete[] notes;
}
//Mètodes consultors
//Mètode per consultar el NIA
//{PRE: Cap}
//{POST: Retorna el NIA de l'estudiant}
long int Estudiant::getNIA() {return NIA;}

//Mètode per consultar el nom
//{PRE: Cap}
//{POST: Retorna el nom de l'estudiant}
char* Estudiant::getNom() {return nom;}
```

Exemmple: Implementació classe Estudiant (cont.)

```
//Mètode per consultar el cognom
//{PRE: Cap}
//{POST: Retorna el cognom de l'estudiant}
char* Estudiant::getCognom() {return cognom;}
//Mètode per consultar la data de naixement
//{PRE: Cap}
//{POST: Retorna la data de naixement de l'estudiant}
long int Estudiant::getDataNaix() {return dataNaix;}

//Mètode per consultar la titulació
//{PRE: Cap}
//{POST: Retorna la titulació de l'estudiant}
char* Estudiant::getTitulacio() {return titulacio;}

//Mètode per consultar la taula de notes
//{PRE: Cap}
//{POST: Retorna la taula de notes de l'estudiant}
double* Estudiant::getNotes() {return notes;}
```

Exemple: Implementació classe Estudiant (cont.)

```
//Mètode per consultar la nota i-èssima
//{PRE: Cap}
//{POST: Retorna la nota i-èssima de l'estudiant}
double Estudiant::getNota_i(int i) {return notes[i];}
//Mètodes modificadores

//Mètode per modificar el NIA
//{PRE: Un enter llarg Id}
//{POST: Assigna Id al NIA de l'estudiant}
long int Estudiant::setNIA(long int Id) {NIA=Id;}

//Mètode per modificar el nom
//{PRE: Una cadena de caràcters N}
//{POST: Assigna N al nom de l'estudiant}
void Estudiant::setNom(char* N){
 //Destruïm el nom vell
 delete [] nom;
 //Adquirim memòria pel nom nou
 nom=new char[strlen(N)+1];
 //Copiem continguts
 strcpy(nom,N);
}
```

Exemple: Implementació classe Estudiant (cont.)

```
//Mètode per modificar el cognom
//{PRE: Una cadena de caràcters C}
//{POST: Assigna C al cognom de l'estudiant}
void Estudiant::setCognom(char* C){
 //Destruïm el cognom vell
 delete [] cognom;
 //Adquirim memòria pel cognom nou
 cognom=new char[strlen(C)+1];
 //Copiem continguts
 strcpy(cognom,C);
}

//Mètode per modificar la data de naixement
//{PRE: Rep un enter llarg data}
//{POST: Assigna la data a la data de naixement de l'estudiant}
void Estudiant::setDataNaix(long int data){dataNaix=data;}
```

Exemmple: Implementació classe Estudiant (cont.)

```
//Mètode per modificar la titulació
//{PRE: Una cadena de caràcters T}
//{POST: Assigna la T a la titulació de l'estudiant}
void Estudiant::setTitulacio(char* T){
 //Destruïm la titulació vella
 delete [] titulacio;
 //Adquirim memòria per la titulacio
 titulacio=new char[strlen(T)+1];
 //Copiem continguts
 strcpy(titulacio,T);
}
//Mètode per modificar la taula de notes
//{PRE: Un real nota i l'índex i de la nota a modificar}
//{POST: Assigna  nota a notes[i] notes de l'estudiant}
void Estudiant::setNota_i(double nota,int i){
   notes[i]=nota;
}
```

Exemmple: Implementació classe Estudiant (cont.)

```
//Operador de sortida
ostream& operator<<(ostream& os,const Estudiant& E){
 os << "Estudiant "<< E.NIA << ' '<< E.nom << ' '<< E.cognom

   << endl;
   os << E.dataNaix << ' '<< E.titulacio << endl;
   os << "Notes: ";
   for (int i=0;i<E.NAssig;i++) os <<E.notes[i] << ' ';
   os << endl;

   return os;
}
```

Edicions UPC
Inici
Contingut
Pàgina 217
Tornar
Pantalla Completa
Tancar
Sortir

Ús de new en l'operador de còpia

Especificació

```
//Constructor de còpia
//{PRE: Rep un objecte de la classe Estudiant}
//{POST: Crea un objecte còpia del objecte donat}
Estudiant(const Estudiant& E);
```

Implementació

```
//Constructor de còpia
//{PRE: Rep un objecte de la classe Estudiant}
//{POST: Crea un objecte còpia del objecte donat}
Estudiant::Estudiant(const Estudiant& E){
    NIA=E.NIA;
    //Adquirim memòria pel nom
    nom=new char[strlen(E.nom)+1];
    //Copiem continguts
    strcpy(nom,E.nom);
```

Exemple: Implementació classe Estudiant (cont.)

```
        //Adquirim memòria pel cognom
        cognom=new char[strlen(E.cognom)+1];
        //Copiem continguts
        strcpy(cognom,E.cognom);

        dataNaix=E.dataNaix;
        //Adquirim memòria per la titulació
        titulacio=new char[strlen(E.titulacio)+1];
        //Copiem continguts
        strcpy(titulacio,E.titulacio);

        NAssig=E.NAssig;
        //Adquirim memòria per la taula de notes
        notes=new double[NAssig];
        //Copiem continguts
        for(int i=0;i<NAssig;i++) notes[i]=E.notes[i];
    }
```

Edicions UPC
Inici
Contingut
Pàgina 219
Tornar
Pantalla Completa
Tancar
Sortir

Ús

```
Estudiant E(...);

//Invocació constructor de còpia
Estudiant E1(E);
```

Ús de new i delete en l'operador d'assignació

Especificació

```
//Operador d'assignació
Estudiant& operator=(const Estudiant& E);
```

Implementació

```
    //Operador d'assignació
    Estudiant& Estudiant::operator=(const Estudiant& E){
if(this!=&E){
        NIA=E.NIA;
        //Destruïm el nom
        delete [] nom;
        //Adquirim memòria pel nom existent
        nom=new char[strlen(E.nom)+1];
        //Copiem continguts
        strcpy(nom,E.nom);
        //Destruïm el cognom existent
        delete [] cognom;
        //Adquirim memòria pel cognom
        cognom=new char[strlen(E.cognom)+1];
        //Copiem continguts
        strcpy(cognom,E.cognom);
```

Implementació (cont.)

```
        dataNaix=E.dataNaix;

        //Destruïm la titulació existent
        delete [] titulacio;
        //Adquirim memòria per la titulació
        titulacio=new char[strlen(E.titulacio)+1];
        //Copiem continguts
        strcpy(titulacio,E.titulacio);

        NAssig=E.NAssig;
        //Destruïm la taula de notes existent
        delete [] notes;
        //Adquirim memòria per la taula de notes
        notes=new double[NAssig];
        //Copiem continguts
        for(int i=0;i<NAssig;i++) notes[i]=E.notes[i];
    }
  return *this;
}
```

Ús

```
Estudiant E(...);

//Invocació constructor de còpia
Estudiant E1(E);

 //Invocació d'operador d'assignació
Estudiant E2=E1;
```

Resum

- Els objectes creats amb **new** resideixen a la zona de memòria anomenada *magatzem lliure* (*free store* o *heap*, en anglès)
- Els objectes de la memòria dinàmica només es poden accedir a través dels apuntadors!

```
Estudiant E2("Pep","Garriga",1984,"E.T. Telecomunicacions");
cout « E2.getNom(); //Correcte
Estudiant *E3 = new Estudiant("Bob","Vallès",1965,"Història");
cout « E3.getNom(); //Error!
cout « E3->getNom(); //Correcte!
```

- Només es destrueixen els objectes en memòria dinàmica!
- Els objectes en memòria dinàmica existeixen fins que no siguin destruïts explícitament!
- L'accés a memòria dinàmica en C++ es fa a través del propi llenguatge, no amb funcions de llibreria
- C++ ha d'assegurar que els tipus dels objectes en memòria dinàmica coincideix amb el dels punters
- C++ ha d'assegurar que els objectes es construeixen i destrueixen

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 10, McGraw-Hill, 2002.
- *Complementàries*:
  - Walter Savitch. *Resolución de problemas con C++*. Capítol 11, Prentice-Hall, 2000.

10. Estructures de Dades: Contenidors i la seva classificació. Introducció a l'STL

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - Contenidors
  - Classificació dels contenidors
  - Contenidors en l'STL. Classificació
  - Iteradors
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 225
Tornar
Pantalla Completa
Tancar
Sortir

Edicions UPC
Inici
Contingut
◀◀ ▶▶
◀ ▶
Pàgina 226
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- A partir de certa complexitat, els programes han de processar informació que ha d'estar estructurada adequadament
- Coneixem l'estructura de dades tipus taula i tipus tupla (*struct*). Sabem com definir i implementar TADs
- Podem necessitar mantenir col·leccions d'objectes del mateix tipus (la taula ja ho és!) que permetin fer operacions eficientment
- Els contenidors són estructures de dades capaç de mantenir una col·lecció d'objectes
- S'implementen de diverses maneres i es cobreix la majoria de les necessitats de programes
- La llibreria STL ofereix diverses implementacions eficients de contenidors

Introducció

- Una estructura de dades és un TAD que defineix un conjunt d'operacions per manipular objectes del TAD.
- Hi ha una família de TADs d'especial interès: els TADs *contenidors* que permeten especificar i implementar diferents estructures de dades.
- Considerarem els contenidors d'ús més comú: els contenidors seqüencials (*vector*, *llistes*, *piles* i *cues*) i un cas de contenidor associatiu: *map*
- Per a cadascun d'aquests TADs veurem:
  - quines operacions proporciona
  - el seu comportament
  - la seva definició proporcionada per l'STL (*Standard Template Library*) una llibreria d'estructures de dades i algorismes de C++
  - exemples del seu ús
  - per últim, també considerarem alguns algorismes bàsics que venen implementats en l'STL per als TADs que veurem

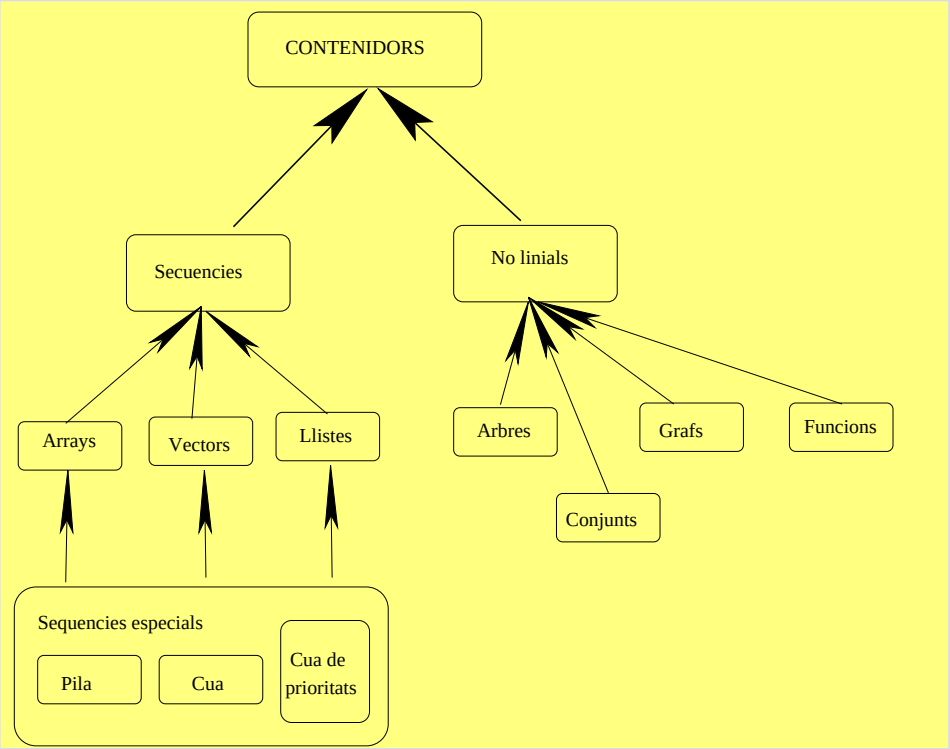
Contenidors: conceptes bàsics

- Contenidor (*container*, en anglès) és qualsevol TAD capaç de contenir una col·lecció d'objectes d'una altra classe (també s'anomenen col·leccions –*collection*, en anglès)
- La seva missió principal és la de proporcionar estructures de dades eficients
- Faciliten/simplifiquen molt la feina del programador ja que no cal fer la gestió de l'espai d'emmagatzemar (es fa a través de constructors, destructor, operacions d'inserció i d'esborrat). A més disposem d'algorismes independents del tipus d'objectes del contenidor
- Exemple de motivació: array d'elements *vs* vector
- Es poden *parametritzar* per oferir genericitat dels objectes que emmagatzemen

Contenidors: conceptes bàsics (cont.)

- Objectius d'ús de contenidors en programació:
  - independència de l'aplicació
  - facilitat d'ús
  - eficiència (temporal i espacial)
- L'API d'un TAD contenidor ha de proporcionar les operacions bàsiques:
  - *inserció* d'un element en el contenidor
  - *supressió* d'un element del contenidor
  - *consulta* dels elements del contenidor
- En general, un contenidor també proporciona les operacions de:
  - *creació* d'objectes (constructor/s del contenidor)
  - *destrucció* d'objectes (destructor del contenidor)
  - constructor de còpia, operador d'assignació, si el contenidor és *buit*, quants elements (*talla*) té el contenidor

Classificació dels contenidors



Edicions UPC
Inici
Contingut
Pàgina 230
Tornar
Pantalla Completa
Tancar
Sortir

Classificació dels contenidors (cont.)

Hi ha cinc famílies principals de contenidors (es basen en models matemàtics):

- *Seqüències*: són contenidors que tenen les característiques d'una seqüència de zero o més elements del mateix tipus $a_0, a_1, \ldots a_{k-1}, a_k, a_{k+1}, \ldots a_{n-1}$, on:
  - a_0 és el *primer element* de la seqüència (inici)
  - a_{n-1} és l'*últim element* (final de la seqüència)
  - per qualsevol element a_k diferent de a_0 i de a_{n-1} hi ha un *anterior* i un *següent*
  - hi ha una forma d'accedir seqüencialment als elements de la seqüència (una regla de successió o d'obtenir el següent).
- Elements organitzats segons on ordre linial
- Exemples d'aquests contenidors són els *arrays*, *piles*, *cues*, *llistes* –també anomenades *estructures de dades linials* (les llibreries d'estructures de dades proporcionen d'altres i diverses variacions)

Classificació dels contenidors (cont.)

- *Arbres*: permeten definir relacions jeràrquiques en un domini d'elements
- *Conjunts*: no hi ha cap ordre entre els elements
- *Funcions*: es defineixen en base de dos dominis especificats A cada element d'un domini li assignen un element d'un altre domini
- *Grafs*: són relacions binàries definides entre elements d'un mateix domini

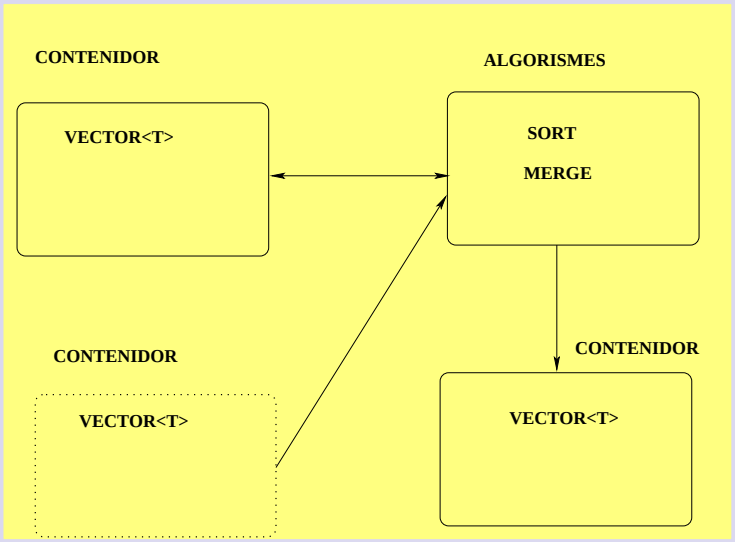
Edicions UPC
Inici
Contingut
Pàgina 232
Tornar
Pantalla Completa
Tancar
Sortir

Introducció a l'STL

- *STL –Standard Template Library–* és una llibreria estàndard d'estructures de dades i algorismes en C++.
- Abstracció, generacitat i eficiència són les característiques més importants de l'STL.
- L'STL consta d'un conjunt de classes que defineixen i implementen les estructures de dades més comunes
- Les implementacions es basen en classes i patrons i fan que siguin implementacions genèriques
- L'STL defineix també un conjunt d'algorismes genèrics sobre les estructures de dades que proporciona
- És de les llibreries més eficients de totes les que hi ha en C++ (i fins i tot d'altres llenguatges de programació)
- *Ens centrarem a saber la interfície de les estructures de dades en l'STL i el seu ús en programes d'aplicacions*

Components de l'STL

- Contenidors: gestió de col·leccions d'objectes
- Iteradors: recrren els objectes d'un contenidor. Connecten contenidors i algorismes
- Algorismes: processen els objectes d'una col·lecció



Contenidors en l'STL

- Un Contenidor en l'STL és una classe capaç de contenir objectes d'una altra classe

  Per exemple, podem tenir un contenidor d'objectes de la classe Pixel o d'objectes de la classe Estudiant

- En un sentit C++ estricte, un contenidor de l'STL és una classe genèrica que es pot instanciar per representar diversos tipus d'objectes

- Hi ha diverses classes de contenidors, es diferencien pel tipus d'operacions que suporten, que s'agrupen en dos grans tipus: *seqüències* i *associacions*.

- Les seqüències emmagatzemen els elements en ordre seqüencial (una successió linial; posició de l'element depen de *quan* i *on* s'ha inserit però no depen del seu valor). Adequades per accés directe y seqüencial.

  – *vector*, *deque*, *list*

  – i *adaptadors*: *stack*, *queue*, *priority_queue*

Contenidors en l'STL (cont.)

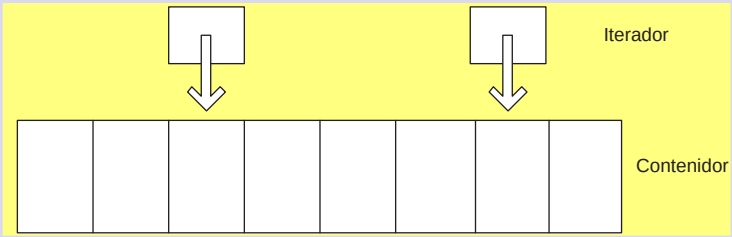
- Un contenidor associatiu emmagatzema objectes de forma ordenada basant-se en una clau. Són adequats per accessos aleatoris mitjançant claus
- La posició actual d'un element depèn només del seu valor segons un criteri d'ordenació dels elements (ordenació automàtica). L'orde d'inserció no importa (no afecta la posició)
- L'STL proporciona quatre tipus diferents (depenent si es permeten claus repetides o no):
  - *map*
  - *multimap*
  - *set*
  - *multiset*

Operacions dels contenidors en l'STL

- Tot contenidor ofereix les següents operacions, independentment del tipus dels objectes que conté (o sigui, són les mateixes per tots els contenidors):
  - constructor per defecte i constructor de còpia
  - operador d'assignació `=`
  - el mètode `size()` que retorna el nombre d'objectes que hi ha al contenidor
  - el mètode `empty()` que retorna `true` si el contenidor està buit i `false` en cas contrari
  - el mètode `begin()` per situar-se (apuntar) al primer element del contenidor i `end()` per situar-se a l'últim element del contenidor (de fet no s'apunta a l'últim element sinó a un element distingit situat després de l'últim)

Iteradors en l'STL

- L'STL proporciona un mecanisme, anomenat *Iterador*, per tal d'accedir (*navegar* per) als elements del contenidor



- Un iterador és una generalització del concepte de punter
- A través d'ells es poden recorrer els objectes del contenidor, cosa que permet la implementació dels algorismes genèrics (amb independència dels tipus d'objectes del contenidor)

Edicions UPC
Inici
Contingut
◀◀ ▶▶
◀ ▶
Pàgina 239
Tornar
Pantalla Completa
Tancar
Sortir

Iteradors en l'STL (cont.)

- Hi ha diferents tipus d'iteradors (ens interessen els d'accés directe, com el dels vectors, i bidireccionals, com els de la llista)
- Per tal d'usar l'iterador, cal declarar-lo segons la sintaxi:

  ```
  contenidor::iterator i;
  ```

- i "connectar-lo" a un objecte contenidor `c` de tipus `contenidor`, per exemple fent:

  ```
  i=c.begin();
  ```

  connectaria l'iterador al principi del contenidor.

Iteradors en l'STL: característiques i operacions

- *Característiques*
  - capacitat de modificar les dades del contenidor. Llavors, poden ser de només *lectura*, només *escriptura*, o només *lectura/escritura*
  - tipus de desplaçament que permeten fer per recórrer el contenidor. Llavors, poden ser d'avanç seqüencial (moure endavant); avanç i retrocés seqüencial (moure endavant i enrera), o d'accés aleatori
- *Operacions*:
  - el constructor per defecte i el de còpia
  - l'operador d'assignació `=`
  - l'operador d'igualtat `==` i desigualtat `!=` per saber si dos iteradors apunten al mateix element
  - l'operador de desreferència `*` per agafar l'objecte apuntat per ell: `t=*I` on `t` és objecte de tipus `T` i `I` un iterador que apunta `t`
  - L'operador d'increment `++` per avançar una posició i de decrement `-` per retrocedir una posició

Exemple

```
#include <...>
#include<vector>
int main(){
    //Declaració d'un contenidor vector de nombres reals
    vector<double> V;
    //Omplim amb valors...
    for (int i=0; i < 10; i++) V.push_back(i*1.1);
    //Per recórrer el vector a través d'un iterador
    //Declarem l'iterador
    vector<double>::iterator I;
    //inicialitzem l'iterador a l'inici del vector
    for (I = V.begin(); I != V.end(); I++) {
     cout << *I << endl;
    }
    return 0;
}
```

Resum

- Els programes de certa complexitat requereixen estructures de dades per tal de consultar i/o manipular la informació eficientment
- Les estructures de dades a més a més han d'oferir interfícies d'operacions comunes pel que fa l'accés, consulta i manipulació de la informació continguda en l'estructura de dades
- Una estructura de dades és un TAD que ofereix una interfície desitjada. Una família de TADs d'interès són els conteniors que són capaços de mantenir una col·lecció d'objectes del mateix tipus

  Una branca important dels contenidors són els seqüencials en què els elements estan estructurats en forma d'una seqüència
- C++ ofreix una llibreia d'estructures de dades eficient, l'STL, que implementa una varietat d'estructures de dades genèriques
- L'STL ofereix els contenidors seqüencials: `vector`, `deque` i `list` i els contenidors associatius: `map` (`multimap`) i `set` (`multiset`)
- Els conteniors de l'STL ofereixen una interfície comuna que facilita el seu ús
- L'STL ofereix el mecanisme d'iterador que permet recórrer els contenidors. La genericitat dels contenidors i els iteradors fan possible una varietat d'algorismes genèrics aplicables als contenidors

Criteris pràctics. Evitar errors

- L'STL cobreix pràcticament totes les necessitats de programar amb estructures de dades i permet obtenir programes eficients
- L'ús de l'STL destaca per la seva facilitat, només cal saber l'API dels contenidors en general i l'API particular de les implementacions concretes proporcionades per la llibreria
- Per tal d'usar correctament les estructures de dades de l'STL, cal conéixer les característiques de cada tipus de contendor i l'eficiència de les seves operacions

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítols 2 i 5, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

11. Estructures de Dades: Contenidors seqüencials. El contenidor `vector`

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - Exemples coneguts de contenidors: taules ordinàries
  - Contenidor seqüencial `vector`
  - Exemple: matrius basades en `vector`
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- En el tema anterior hem vist el concepte de contenidor
- Una família important de contenidors són els seqüencials
- `vector` és una de les implementacions que ofereix l'STL

Exemples coneguts de contenidors seqüencials

- Arrays ordinàries de C++: `T a[MAX];`
  - accés directe a una seqüència d'elements de tipus `T`
  - tamany fix `MAX`
- Les cadenes de caràcters són un altre exemple

Vector: `vector<T>`

- Permet accés directe a partir de la posició (com les arrays)
- Internament és un array d'elements de tipus T. Podem fer les mateixes operacions d'una taula ordinària, a més de créixer dinàmicament

Vector: `vector<T>` (cont.)

- Permet inserir al final en temps constant. Les insercions en qualsevol altra posició és fan en temps linial
- Es pot fer créixer dinàmicament (bo per situacions quan no sabem a priori el nombre d'elements a emmagatzemar o el sabem aproximadament)

Taula comparativa de la complexitat de les operacions

	array	**vector**
accedir al primer element	**constant**	**constant**
accedir a l'últim element	**constant**	**constant**
accedir a un element intermig	**constant**	**constant**
inserir/borrar al principi	no	linial
inserir/borrar al final	no	**constant**
inserir/borrar en mig	no	linial

Operacions bàsiques

Constructors

- *Constructor per defecte*: `vector<T> v;`

  Crea un vector buit d'elements de tipus T. Per ex.: `vector<double> Notes;` construeix un vector de reals buit. Observeu com s'indica el tipus de dades (`double` en l'exemple)

- *Constructor que especifica el tamany*: `vector<T> v(n);`

  Crea un vector de tamany `n` (per cadascun dels `n` elements es cridat el constructor per defecte del tipus)

- *Constructor que inicialitza un cert nombre d'elements a un cert valor especificat*: `vector<T> v(n,el);`

  Crea un vector de tamany `n`, inicialitzat amb n còpies de l'element `el`. Per ex.: `vector<double> Notes1(35,0.0);` construeix un vector amb 35 elements inicialitzats tots a 0

Constructors (cont.)

- *Constructor de còpia*: `vector<T> v2(v1);`

  Crea una còpia v2 del vector v1 (es copien tots els elements). Per ex.: `vector<double> Notes2(Notes1);`

Operacions consultores (les de tots els contenidors)

- `v.size()` retorna el nombre d'elements
- `v.empty()` retorna si el contenidor és buit
- `v.max_size()` retorna el nombre màxim possible d'elements
- `v.capacity()` retorna el nombre màxim possible d'elements sense re-assignar memòria

Assignació

La usual dels contenidors seqüencials.

Accés als elements

- `v[idx]`, operador d'accés als elements `[]` tant per llegir com per escriure a una posició del vector (recordeu l'operador d'accés a les taules). Per exemple, `Notes1[0]=7.5; cout << Notes1[0];`
- El mètode `front()` per consultar el primer element del vector. Per exemple, `cout << Notes1.front();` retornaria `7.5`
- El mètode `back()` per consultar l'últim element del vector. Per exemple, `cout << Notes1.back();` retornaria `0`

Edicions UPC
Inici
Contingut
Pàgina 252
Tornar
Pantalla Completa
Tancar
Sortir

Exemple

```
//Per usar el contenidor vector cal incloure'l
#include<vector>
int main(){
  vector<double> Notes;
  cout << "Notes te tamany: "<< Notes.size() << endl;

  vector<double> Notes1(35,0.0);
  cout << "Notes1 te tamany: "<< Notes1.size() <<endl;
  cout << "Tercer element de Notes1= << Notes1[2]<< endl;

  vector<double> Notes2(Notes1);
  cout << "Notes2 te tamany: << Notes2.size()<< endl;

  Notes1[0]=6.7;
 cout<<"L'element retornat amb front() es: "<<Notes1.front()<<endl;
 cout<<"L'element retornat amb back() es: "<<Notes1.back()<<endl;
  return 0;
 }
```

Inserció d'elements

- `push_back(elem)` afegeix `elem` al final del vector. Per exemple, `Notes1.push_back(6.0);`
- Hi ha definit el mètode `insert()`, l'ús del qual requereix l'iterador (ho veurem més endavant)

Esborrat d'elements

- El mètode `pop_back()` elimina l'últim element del vector (no el retorna!). Per exemple, `Notes1.pop_back();`
- El mètode `clear()` borra tots els elements del vector (deixa el contenidor buit!)
- Hi ha definit el mètode `erase()`, l'ús del qual requereix l'iterador (ho veurem més endavant)

Redimensionar el vector

- El mètode `resize(n)` permet redimensionar el vector al tamany proporcionat `n` (que pot ser més gran o més petit que el tamany actual del vector!). Per exemple, `Notes1.resize(60);` (recordem que el mètode `size()` retorna el tamany del vector)

Iteradors sobre `vector`

- L'iterador definit sobre el vector permet recórrer-lo còmodament (no cal saber el tamany del vector!) i permet fer operacions d'inserció i esborrat
- Un iterador sobre `vector` es declara com segueix:

```
vector<T>::iterator nom_iterador;
```

on `T` és el tipus de dades dels elements del vector

Iteradors sobre `vector` (cont.)

- Com es connecta l'iterador definit amb un vector? Suposem que volem un iterador per al vector `Notes1` (vegeu abans)
  - primer declarem l'iterador

```
vector<double>::iterator I;
```

  - segon, connectem `I` amb el vector `Notes1` fent: `I=Notes1.begin();` que situa l'iterador al primer element del vector
  - ara ja podem recórrer el vector `Notes1` fent servir l'iterador `I`:

```
vector<double>::iterator I;
 I=Notes1.begin();
 while (I != Notes1.end()){//recorregut
   cout << *(I) << "\t";
   I++;
 }
```

Les operacions `insert()` i `erase()`

- El mètode `v.insert(pos,elem)` insereix `elem` a la posició apuntada per `pos` (l'iterador ara apunta sobre l'element afegit; retorna la posició del nou element)
  - per exemple, `Notes1.insert(Notes1.begin(), 7.8);`
  - nota: recordar que els elements del vector ocupen una zona contígua de memòria
- El mètode `v.erase(pos)` elimina l'element del vector apuntat per l'iterador `pos`. Retorna la posició del següent element
  - per exemple, `Notes1.erase(Notes1.begin());`

Exemple d'ús: vector de pixels

```
#include <Pixel.hpp>
#include<vector>
bool sortir(char resposta){
    return (resposta=='S'||resposta == 's');
int main(){
   vector<Pixel> VP;
   double R,G,B;
   char resposta='N';
    //omplim el vector amb pixels
     while(!sortir(resposta)){
       cout << "Introdueix els valors R, G, B:";
       cin >> R >> G >> B;
       Pixel P(R,G,B);
       VP.push_back(P);
       cout << "Sortir? (S/N)"; cin>>resposta;
     }
     cout << "Quants pixels s'han introduit?: "<< VP.size() << endl;
```

Exemple d'ús: vector de pixels (cont.)

```
    //calculem la mitjana de cada color
    vector<Pixel>::iterator it = VP.begin();
    double sumaR=0.0; double sumaG=0.0;double sumaB=0.0;
     while (it!=VP.end()){
       sumaR=sumaR+(*it).getRed();
       sumaG=sumaG+(*it).getGreen();
       sumaB=sumaB+(*it).getBlue();
       it++;
     }
     if (VP.size()>0){
        cout << "Mitjana Red: "<< sumaR/VP.size() << endl;
        cout << "Mitjana Green: "<< sumaG/VP.size() << endl;
        cout << "Mitjana Blue: << sumaB/VP.size() << endl;
     }
     else cout << "No s'han introduit pixels"<< endl;
  return 0;
}
```

Edicions UPC
Inici
Contingut
◀◀ ▶▶
◀ ▶
Pàgina 259
Tornar
Pantalla Completa
Tancar
Sortir

Nota: a l'hora d'utilitzar el contenidor vector (i altres contenidors) cal que el tipus d'objectes T proporcioni:

- constructor per defecte
- constructor de còpia
- operador d'assignació

Exemple: matrius usant `vector`

- Podem usar el contenidor `vector` per construir taules bidimensionals (un vector els elements del qual són vectors)

```
//Programa que declara i usa matrius usant el tipus vector
#include<vector>
const int N=...;
typedef vector<double> tFila;
double suma(vector<tFila> M,int N){
    double s=0.0;
    for (int i=0;i<N;i++)
        for (int j=0;j<N;j++)
            s=s+M[i][j];
   return s;
}
```

Exemple: matrius usant `vector` (cont.)

```
int main(){
    vector<tFila> M(N,tFila(20));
    for (int i=0;i<N;i++)
        for (int j=0;j<N;j++)
            M[i][j]=i+j;
    cout << " Suma de la matriu: "<< suma(M,N) << endl;
    return 0;
}
```

Resum

- `vector` és un contenidor seqüencial de l'STL, que es pot instanciar per a qualsevol tipus (ja sigui bàsic com qualevol altre definit per l'usuari)
- Amplia el concepte de l'estructura de dades array oferint una interfície pròpia i permet créixer l'estructura dinàmicament. Es pot redimensionar
- Permet accés directe als elements amb l'operador usual `[]` de les arrays i accés a través de l'iterador
- Permet inserir elements pel final en temps (gairebé) constant (estructura que creix pel final)
- Altra informació, com ara el tamany, es pot consultar amb els mètodes proporcionats pels contenidors en general

Criteris pràctics. Evitar errors

- El contenidor `vector` és adequat per casos en què es fan moltes consultes a les dades i tenim prou amb insercions pel final de l'estructura
- Per tal d'accedir a una posició del vector amb l'operador `[]` cal haver declarat el vector amb un tamany inicial
- Com que els elements del vector s'emmagatzemen en una zona contígua de la memòria, l'aritmètica dels punters és aplicable als iteradors (p.ex. podem accedir a la posició I+5, essent I un iterador del vector)

```
vector<int>V(20);
V[5]=10; vector<int>::iterator I=V.begin();
cout << *(I+5);
```

- S'ha de vigilar de no sortir fora del rang del vector

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítols 5 i 6, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

Edicions UPC
Inici
Contingut
Pàgina 265
Tornar
Pantalla Completa
Tancar
Sortir

12. Estructures de Dades: Estructura de dades llista. El contenidor `list`

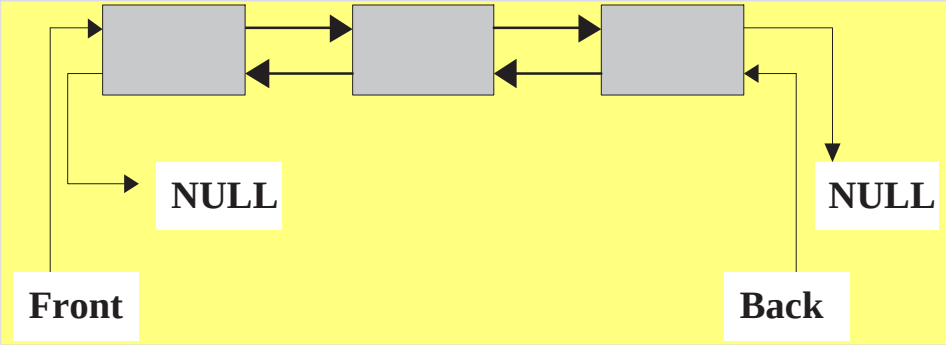
- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - llista / `list` en l'STL
  - operacions bàsiques
  - exemples d'ús
  - `list` *versus* `vector`
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- En la classe anterior hem vist l'estructura de dades `vector`
- Destaca per: proporcionar accés directe als elements! però afegir/treure elements és costós (tret del de l'última posició)
- El vector està estructurat internament com un array, un bloc de cel·les contígües de memòria
- La llista és una altra estructura de dades seqüencial molt important
- La llista es pot implementar de diverses maneres, l'STL proporciona una llista doble enllaçada
- La llista perd l'habilitat dels vectors de donar accés directe als elements; en canvi, però permet fer les operacions d'afegir/treure elements de manera eficient
- L'organització de la llista en memòria permet un millor ús de la memòria en no ser organitzat en un bloc de cel·les contígues de la memòria com el vector

Concepte de llista / `list` en l'STL

- Estructura d'una llista:



- Contenidor seqüencial implementat com a llista doblement encadenada
- Proporciona una estructura de dades eficient que permet inserir i esborrar en temps constant en qualsevol posició de la llista (incloent els dos extrems de la llista!) Raó?
- Proporciona iterador (bidireccional) per accedir als elements i "navegar" en totes dues direccions per la llista

Taula comparativa de la complexitat de les operacions (vector vs list)

	list	vector
accedir al primer element	constant	constant
accedir a l'últim element	constant	constant
accedir a un element intermig	linial	constant
inserir/borrar al principi	constant	linial
inserir/borrar al final	constant	'constant'
inserir/borrar en mig	constant	linial

Com es pot observar en aquesta taula:

- La llista proporciona accés en temps linial a qualsevol element de la llista que no sigui el primer o l'últim

  Accedir, però, a un element de la llista requerix navegar per la llista per arribar-hi a l'element desitjat, cosa que fa que sigui lent!

- Les insercions es fan en temps constant en qualsevol posició de la llista
- Els esborrats es fan en temps constant en qualsevol posició de la llista

Edicions UPC
Inici
Contingut
Pàgina 269
Tornar
Pantalla Completa
Tancar
Sortir

Operacions bàsiques

Constructors

- *Constructor per defecte*: `lis<T> L;`

  Crea una llista buida d'elements de tipus T. Per ex.: `list<string> Torres;` construeix una llista buida d'strings (noms de Torres). Observeu com s'indica el tipus de dades (`string` en aquest cas)

- *Constructor que especifica el tamany*: `list<T> L(n);`

  Crea una llista amb `n` elements (per cadscún dels `n` elements és cridat el constructor per defecte del tipus)

- *Constructor que inicialitza un cert nombre d'elements a un cert valor especificat*: `list<T> L(n,el);`

  Crea una llista de `n` elements, inicialitzat amb `n` còpies de l'element `el`. Per exemple: `list<string> Torres(10,"Eiffel");` construeix una llista amb 10 elements inicialitzats tots a "Eiffel"

Operacions bàsiques (cont.)

Constructors (cont.)

- *Constructor de còpia*: `list<T> L2(L1);`

  Crea una còpia L2 de la llista L1 (es copien tots els elements). Per exemple: `list<string> Torres1(Torres);`

Operacions consultores

Les usuals de tots els contenidors respecte al tamany i les comparacions:

- `l.size()` retorna el nombre d'elements
- `l.empty()` retorna si el contenidor és buit
- `l.max_size()` retorna el nombre màxim possible d'elements

Edicions UPC
Inici
Contingut
Pàgina 271
Tornar
Pantalla Completa
Tancar
Sortir

Operaors relacionals

Quant a les comparacions, a part de == i != hi ha implementats els <, <=, >, >=, que estableixen la comparació segons un ordre lexicogràfic (com les cadenes de caràcters)

Assignació

L'usual dels contenidors

Accés als elements

Com que la llista no propociona accés directe, només es pot accedir directament els elements `front` i `back`

- El mètode `front()` per accedir al primer element de la llista. Per exemple, `cout << Torres.front();` retornaria `Eiffel` (no es comprova si existeix tal element!)

Accés als elements (cont.)

- El mètode `back()` per accedir a l'últim element de la llista. Per exemple, `cout << Torres.back();` retornaria `Eiffel` (no es comprova si existeix tal element!)

Iteradors sobre `list`

- Hi ha iterador bidireccional, que es pot incrementar i decrementar (no hi ha iterador d'accés directe)
- Un iterador sobre `list` es declara com segueix:

  ```
  list<T>::iterator nom_iterador;
  ```

  on `T` és el tipus de dades dels elements de la llista
- L'iterador definit permet recórrer còmodament la llista en tots dos sentits i també permet fer operacions d'inserció i esborrat

Edicions UPC
Inici
Contingut
Pàgina 273
Tornar
Pantalla Completa
Tancar
Sortir

Iteradors sobre `list` (cont.)

- Com es connecta l'iterador definit amb la llista? Suposem que volem un iterador per la llista `Torres`
  - primer declarem l'iterador
    `list<string>::iterator I;`
  - segon, connectem `I` amb la llista `Torres` fent: `I=Torres.begin();` que situa l'iterador al primer element de la llista
- ara ja podem recórrer la llista `Torres` fent servir l'iterador `I`:

```
list<string>::iterator I;
  I=Torres.begin();
  while (I != Torres.end()){//recorregut
    cout << *I << "\t";
    I++;
  }
```

Iteradors sobre `list` (cont.)

Alternativament, podem recórrer la llista de dreta a esquerra:

```
list<string>::iterator it=Torres.end();
  while(it!=Torres.begin()){
     it--;
     cout << *(it)<< "\t";
  }
```

Nota: `end()` apunta a un element fantasma (no vàlid a efectes de tractament), o sigui a la posició després de l'últim element!

Inserció d'elements

- *Inserció*: al principi, al final i en qualsevol posició.
  - `push_front(valor)` afegeix `valor` al principi de la llista. Per exemple, `Torres.push_front("Torres de Hannoi");`
  - `push_back(valor)` afegeix `valor` al final de la llista. Per exemple, `Torres.push_back("Torre de l'Aigua");`
  - el mètode `insert(pos,valor)` insereix valor a la posició apuntada per `pos` (l'iterador ara apunta sobre l'element afegit), per exemple, `Torres.insert(it, "Torre Picasso");`

Esborrat d'elements

- *Esborrat*: del primer, de l'últim i en qualsevol posició.
  - el mètode `pop_front()`. Elimina el primer element de la llista (no el retorna!). Per exemple, `Torres.pop_front();`
  - el mètode `pop_back()`. Elimina l'últim element de la llista (no el retorna!). Per exemple, `Torres.pop_back();`
  - el mètode `erase(pos)`. Elimina l'element de la llista apuntat per l'iterador `pos` (l'iterador apunta al següent element), per exemple, `Torres.erase(pos);`
- El mètode `remove(valor)`. Elimina totes les ocurrències del `valor` de la llista
- El mètode `unique()`. Elimina els elements duplicats (consecutius!). Opera amb llistes ordenades
- El mètode `clear()`. Esborra tots els elements del vector (deixa el contenidor buit!)

Altres operacions

- `l.sort()`. Ordena els elements de la llista `l` segons l'ordre <. En cas e llista elements no bàsics i de tipus string, cal que el tipus (la classe) d'elements tingui definit l'operador de comparació <
- `l1.merge(l2)`. Mescla les dues llistes ordenades `l1` i `l2` i deixa el resultat en `l1`
- `l.reverse()`. Posa els elements de la llista `l` en l'ordre al revés
- `l.size()`. Consultar el tamany (nombre d'elements de la llista (mètode proporcionat per tot contenidor)
- `l.empty()`. Consultar si la llista està buida. Cal aplicar aquest mètode abans de fer consultes i esborrats d'elements de la llista

Exemple d'ús: llista de noms d'imatges

```
//Per usar list cal incloure'l
#include<list>

//Mètode per mostrar la llista
void mostraLlista(list<string> & l){
    list<string>::iterator it = l.begin();
    while (it!= l.end()){
        cout << *it << "\t";
        it++;
    }
    cout << endl;
}

int main(){
  //Declarem la llista
  list<string> NomsImatges;
  //Llegim noms d'imatges i els posem a la llista, Fi per acabar
  string nom;
  cin >> nom;
```

Exemple d'ús: llista de noms d'imatges (cont.)

```
while(nom!="Fi"){
        NomsImatges.push_back(nom); // afegim nom pel final
        cin >> nom;
}
cout << "Quants noms d'imatges s'han introduit?: "
                << NomsImatges.size() << endl;
// Imprimim la llista de noms introduïts
mostraLlista(NomsImatges);
NomsImatges.reverse();
cout << "La llista reverse:"<< endl;
mostraLlista(NomsImatges);
//Ordenem la llista
NomsImatges.sort();
cout << "La llista ordenada:"<< endl;
mostraLlista(NomsImatges);
//Apliquem unique()
NomsImatges.unique();
//Mostrem la llista de nou
mostraLlista(NomsImatges);
```

Exemple d'ús: llista de noms d'imatges (cont.)

```
  //Fem una operació d'esborrat
  list<string> :: iterator it= NomsImatges.end();
  it--;
  NomsImatges.erase(it);
  //Mostrem la llista de nou
  mostraLlista(NomsImatges);
   return 0;
}
```

`list` *versus* `vector`

- Tant `vector` com `list` són estructures linials amb similituds. Quina escollir?
- Si volem accés en temps constant, `vector` és preferible
- En cas de saber aproximadament el tamany de l'estructura (nombre d'elements a emmagatzemar) `vector` és preferible
- Si les operacions que predominen són les insercions / esborrats, `list` és preferible

Nota: a l'hora d'utilitzar el contenidor llista (i altres contenidors) cal que el tipus d'objectes T proporcioni:

- constructor per defecte
- constructor de còpia
- operador d'assignació

Resum

- `list` és un contenidor seqüencial de l'STL que es pot instanciar per a qualsevol tipus (ja sigui bàsic com qualsevol altra definit per l'usuari)
- És una estructura de dades dinàmica implementada com llista doblement enllaçada
- Permet accés directe al primer i últim element però no hi ha accés directe pels elements de qualsevol posició
- Destaca per l'eficiència de les operacions d'inserció i esborrat en qualsevol posició: es fan en temps constant
- Ofereix iterador bidireccional que permet recórrer la llista en tots dos sentits
- Altra informació, com ara el tamany, es pot consultar amb els mètodes proporcionats pels contenidors en general

Criteris pràctics. Evitar errors

- El contenidor list és adequat per casos en què es fan moltes modificacions de les dades
- A diferència del `vector`, els elements del qual s'emmagatzemen en una zona contígua de la memòria, l'iterador de la llista només proporciona les operacions `++` i `--` (no podem accedir a la posició I+5, essent I un iterador de la llista!)
- En utilitzar l'iterador de la llista en el sentit de dreta a esquerra, cal fer primer `I--` per tal de situar-se a l'últim element vàlid de la llista
- Per tal d'utilitzar certs mètodes de la llista com ara el `sort`, cal que el tipus dels elements de la llista tingui implemenat l'operador relacional `<`

Edicions UPC
Inici
Contingut
Pàgina 284
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítols 5 i 6, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

13. Estructures de Dades: Estructura de dades Pila. Adaptadors de l'STL. Adaptador `stack`

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - concepte d'estructura de dades pila / `stack` en l'STL
  - operacions bàsiques
  - exemples d'ús
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

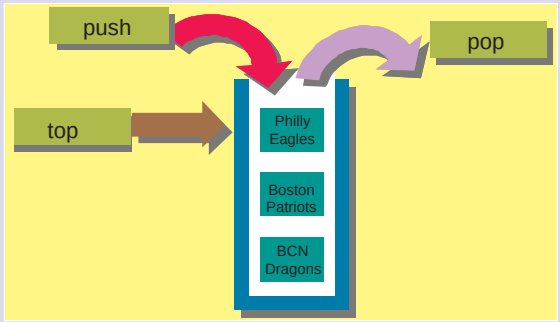
Edicions UPC
Inici
Contingut
Pàgina 285
Tornar
Pantalla Completa
Tancar
Sortir

Motivació. Relació amb el tema anterior

- En els temes anteriors hem vist els contenidors seqüencials `vector` i `list`
- En moltes aplicacions, però, podem necessitar contenidors seqüencials que ofereixin una interfície que s'adapti bé per algunes aplicacions
- A més, és força apropiat tenir contenidors que ofereixen una interfície senzilla i que s'assembla a models reals
- Aquests contenidors són, entre d'altres, les estructures de dades Pila (`stack`) i Cua (`queue`)
- Com que són contenidors "especials", en l'STL aquests s'anomenen *adaptadors*
- Els adaptadors es construeixen sobre els contenidors seqüencials com `vector` o `list` (són com una mena d'envoltoris (*wrapper*, en anglès) d'aquests contenidors a fi d'oferir una interfície particular)

Concepte d'estructura de dades pila

- Les operacions d'inserció, esborrat i consulta es poden fer *només* pel "final" de la seqüència, anomenat *cim* (*top*, en anglès) de la pila
- Estructura de dades seqüencial LIFO (Last In – First Out): l'últim element afegit és el primer a sortir
- Operacions: terminologia pròpia
  - insertar ⟶ apilar (*push*, en anglès)
  - eliminar ⟶ desapilar (*pop*, en anglès)
  - consultar ⟶ cim (*top*, en anglès)



Exemples

- Pila de canvis recents d'una imatge
- Modificacions recents d'un text
- Capgiar un text llegit pel canal d'entrada
- Pila d'exàmens
- Pila d'execució d'un programa (crida de funcions)
- Pila d'adreces web visitades recentment
- En llenguatges de programació (per exemple en programes per fer tractament d'expressions matemàtiques)

L'adaptador `stack` de l'STL

Una pila en l'STL (`stack`) és un adaptador sobre un `vector`, `list` (o `deque`), que només té les operacions pròpies de la pila

Per tal de definir una pila en l'STL cal indicar el contenidor seqüencial base, a més d'incloure el fitxer de capçalera `stack` a través de la directiva `#include<stack>`.

Constructors

- `stack<T> s;` constructor d'una pila buida basada en el contenidor per defecte (`dequeu`). Per exemple `stack<string> adreces_web;`
- `stack<T, vector<T> > s;` constructor d'una pila basada en vectors. Per exemple `stack<string,vector<string> > adreces_web;`
- `stack<T, list<T> > s;` constructor d'una pila basada en llistes. Per exemple `stack<string,list<string> > adreces_web;`
- Constructor de còpia: com usualment. `stack<T> s1(s);` construeix `s1` còpia de `s`.

Operacions bàsiques

- `push(valor);`

  Insereix `valor` (del tipus dels elements de la pila) al cim de la pila. Per exemple `adreces_web.push("www.upc.edu");`

- `pop();`

  Elimina el cim (darrer element) de la pila (no el retorna!). Per exemple `adreces_web.pop();`

- `valor=top();`

  Funció de *consulta* que *retorna* (no el treu!) l'element cim (referència d'ell) de la pila. Per exemple, `string adreca=adreces_web.top();`

Altres operacions

- `s.size()` per consultar el tamany (nombre d'elements de la pila `s`).
- `s.empty()` per consultar si la pila està buida
- operador d'assignació = (operador proporcionat per tot contenidor).

Edicions UPC
Inici
Contingut
Pàgina 292
Tornar
Pantalla Completa
Tancar
Sortir

Observacions

- les operacions de l'API de la pila són independents del contenidor base usat!
- no hi ha iteradors!
- cal deixar un espai enmig `> >` quan es declara el contenidor base.
- per buidar una pila cal eliminar reiteradament el cim de la pila
- el comportament de les operacions `top()` i `pop()` no està definit per una pila buida. Cal consultar l'estat de la pila amb `empty()` o `size()` abans d'aplicar les operacions de consulta i esborrat
- hi ha definit els operadors bàsics dels contenidors (assignació, operadors relacionals)

Exemple: pila per mantenir les adreces web visitades recentment

```
#include<string>
#include<stack>
#include<vector>
string opcio(){
       cout << "Opcio: Quantes, Nova, Ultima, Enrera"<<endl;
       string opcio;
       cin >> opcio;
      return opcio;
}
int main() {
  stack<string,vector<string> > adreces;
  bool sortir=false;
  while (!sortir){
          string op=opcio();
          if (op=="Nova"){///Agegim una adreça a la pila
            cout << "Nova adreca visitada: ";
            string adreca;
```

Exemple: Pila per mantenir les adreces web vistitades recentment (cont.)

```
    cin >> adreca;
    adreces.push(adreca);
   }
  else if (op=="Ultima"){//Consultem l'última adreça visitada
     if (!adreces.empty()){
       cout << "Ultima adreca visitada: "<< adreces.top() <<endl;
     }
     else cout << "No hi ha adreces!"<<endl;
     }
  else if (op=="Enrera"){       //Ens movem a l'adreça anterior
       if (!adreces.empty()){ //mirem que pila no estigui buida
         adreces.pop();//eliminem el cim
         string adreca=adreces.top();
         cout << "Enrera...: "<< adreca <<endl;
       }
       else cout << "No hi ha adreces!"<<endl;
  }
```

Exemple: Pila per mantenir les adreces web vistitades recentment (cont.)

```
  else if (op=="Quantes")  //Consultem quantes adreces s'han visitat
    cout << "Hi ha "<< adreces.size() << " adreces visitades"<< endl;
     cout << "Sortir (S/N)?";
     char resposta; cin >> resposta;
     sortir=(resposta=='S'||resposta=='s');
   }
   return 0;
}
```

Resum

- Pila és una estructura de dades bàsica i de molta utilitat en programació
- Prové de models reals (“pila d’objectes”) cosa que fa que tingui una interfície pròpia
- La peculiaritat de la pila és que es té accés només a un extrem de la seqüència, a l’element final anomenat cim de la pila
- Aquesta peculiaritat fa que la pila es conegui com l’estructura LIFO (Last In First Out), és a dir, l’últim en entrar és el primer a sortir

  La pila és d’aplicació quan s’ha de “donar servei” segons l’ordre LIFO, com ara tractament de canvis recents d’un text per tal de poder desfer-los
- Tot i ser una estructura de dades linial, en l’STL és un *adaptador* construït sobre vector, dequeu o list

Criteris pràctics. Evitar errors

- Per consultar els elements de la pila (per exemple, per fer un llistat) cal aplicar iteradament, en aquest ordre, les operacions: `top()` (consulta) i `pop()` (elimina)
- En aplicar `top()` i `pop()` es buida la pila per la qual cosa, per no perdre els elements cal fer abans una còpia de la pila a través del constructor de còpia
- Abans d'aplicar les operacions de consulta i esborrat cal comprovar (usant el mètode `empty()`) que la pila no estigui buida
- D'altra banda, pel que fa les insercions, no hi ha límit, tret de la capacitat física de la memòria, ja que la pila en l'STL és una estructura de dades dinàmica

Edicions UPC
Inici
Contingut
Pàgina 298
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítol 10, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

14. Estructures de Dades: Estructura de dades Cua. Adaptador `queue`

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - concepte d'estructura de dades cua / `queue` en l'STL
  - operacions bàsiques
  - exemples d'ús
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 299
Tornar
Pantalla Completa
Tancar
Sortir

Edicions UPC
Inici
Contingut
Pàgina 300
Tornar
Pantalla Completa
Tancar
Sortir

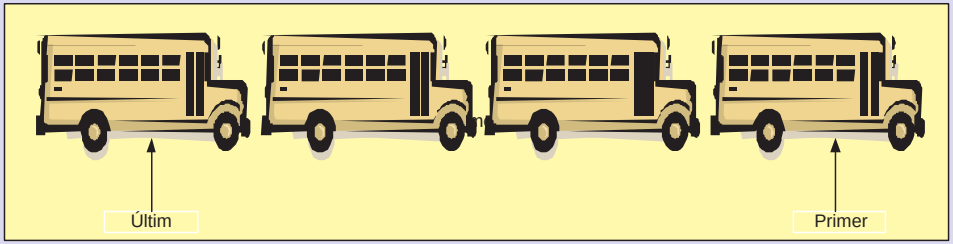
Motivació. Relació amb el tema anterior

- L'estructura de dades cua (`queue`, en anglès) és una altra estructura de dades fonamental en programació
- En moltes aplicacions cal tractar les dades conforme l'ordre d'arribada. Per exemple, cal donar servei a les peticions d'impressió dels usuaris segons l'ordre d'arribada de les peticions
- Així, l'estructura de dades cua prové de models reals, cosa per la qual ofereix una interfície pròpia
- La seva peculiaritat rau en el fet que sols es té accés als dos extrems de la seqüència: primer i últim element
- En l'STL la cua és un *adaptador* que usa per sota un contenidor seqüencial (`dequeu` o `list`). És un embolcall a un contenidor seqüencial per oferir la interfície de la cua

Edicions UPC
Inici
Contingut
Pàgina 301
Tornar
Pantalla Completa
Tancar
Sortir

Concepte d'estructura de dades cua

- L'operació d'inserció es fa pel final de la seqüència
- Les operacions d'esborrat i consulta es fan per l'inici de la seqüència
- Estructura de dades seqüencial FIFO (First In – First Out): el primer element entrat és el primer a sortir (esborrat i consulta)



Operacions: terminologia pròpia

- insertar $\longrightarrow$ encuar (*push*, en anglès)
- eliminar $\longrightarrow$ desencuar (*pop*, en anglès)
- consultar $\longrightarrow$ primer (*front*, en anglès)

Exemples

- Cua d'impressió de documents
- Cua d'un peatge
- Cua de trucades a 012
- Cua de processos en un ordenador

L'adaptador `queue`

Una cua de l'STL (`queue`) és un adaptador sobre un contenidor list o deque (no vector!), que només té les operacions pròpies de la cua

Per tal de definir una cua en l'STL s'ha d'indicar el contenidor seqüencial base (a més, cal incloure el fitxer de capçalera `queue` amb la directiva `#include<queue>`)

Constructors

- `queue<T> q;`

  Constructor d'una cua basada en el contenidor per defecte. Per exemple, `queue<int> CuaIdDocsImpressio;` declara una cua per guardar els identificadors de documents d'impressió que s'envien a una impressora

Constructors (cont.)

- `queue<T, list<T> > q;`

  Constructor d'una cua basada en llistes. Per exemple, `queue<Document,list<Document> > CuaDocsImpressio;` declara una cua per guardar objectes de la classe `Document` (informació d'un document enviat a la impressora)

Operacions bàsiques

- `push(valor);` afegeix un element al final de la cua
- `pop();` elimina el primer element de la cua
- `valor=back();` retorna el darrer element de la cua (no l'esborra!). És funció de consulta
- `valor=front();` retorna el primer element de la cua (no l'esborra!). És funció de consulta

Edicions UPC
Inici
Contingut
Pàgina 305
Tornar
Pantalla Completa
Tancar
Sortir

Altres operacions

- `c.size()` per consultar el tamany (nombre d'elements de la cua `c`)
- `c.empty()` per consultar si la cua `c` està buida
- operador d'assignació =

Observacions

- Les operacions de l'API de la cua són independents del contenidor base usat!
- No hi ha iteradors!
- Cal deixar un espai enmig `> >` quan es declara el contenidor base
- Per buidar una cua cal eliminar reiteradament el front de la cua
- El comportament de les operacions `front()`, `back()` i `pop()` no està definit per una cua buida. Cal consultar l'estat de la cua amb `empty()` o `size()` abans d'aplicar les operacions de consulta i esborrat

Exemple: Programa per simular la cua d'una impressora

```
//Classe Document per gestionar la informació d'un document
//d'impressió
class Document{
   int Id;  //Identificador del document
   string titol; //Títol del document
   string usuari; //Usuari (propietari) qui ha enviat el document
 public:
   // constructor per defecte
   Document(){Id=0;}
   // constructor amb paràmetres
   Document(int unId,string unTitol,string unUsuari){
      Id=unId;
      titol=unTitol;
      usuari=unUsuari;
   }
```

Exemple: Programa per simular la cua d'una impressora (cont.)

```
    //constructor de còpia
    Document(const Document& D){
    Id=D.Id;
    titol=D.titol;
    usuari=D.usuari;
    }
    //Mètodes get
    int getId(){ return Id;}
    string getTitol(){return titol;}
    string getUsuari(){return usuari;}
    //Mètodes set
    void setId(int unId){Id=unId;}
    void setTitol(string unTitol){titol=unTitol;}
    void setUsuari(string unUsuari){usuari=unUsuari;}
    //Operador d'assignació
    Document& operator=(const Document& D) {
      if (this!=&D){
         Id = D.Id; titol=D.titol; usuari=D.usuari; }
      return *this;
    }
};
```

Exemple: Programa per simular la cua d'una impressora (cont.)

```
#include<...>
#include<list>
#include<queue>
string opcio(){
  cout << "Opcio: Quants, Nou, Imprimir, Llistat"<<endl;
  string opcio; cin >> opcio;
  return opcio;
}
int main(){
  //Declarem la cua dels documents
  queue<Document,list<Document> > CuaDocsImpressio;
  bool sortir=false;
  while (!sortir){
     string op=opcio();
     if (op=="Nou"){
        cout << "Nou document: ";
```

Exemple: Programa per simular la cua d'una impressora (cont.)

```
    int id; cin >>id;
    string titol; cin >> titol;
    string usuari; cin>>usuari;
    Document D(id,titol,usuari);
    CuaDocsImpressio.push(D);
  }
  else if (op=="Imprimir"){
      if (!CuaDocsImpressio.empty()){
        Document D=CuaDocsImpressio.front();
        cout << "S'imprimeix...: "<< D.getId() << "\t"<< D.getTitol()
                                 << "\t"<< D.getUsuari() <<endl;
        CuaDocsImpressio.pop();
      }
      else cout << "No hi ha document d'impressio!"<< endl;
  }
```

Exemple: Programa per simular la cua d'una impressora (cont.)

```
else if (op=="Quants") {
    cout << "Hi ha "<< CuaDocsImpressio.size() <<
            " documents en la cua de la impressora"<< endl;
}
else if (op=="Llistat"){
    if (CuaDocsImpressio.empty()) cout << "Cua impressio buida!";
    else {
        cout << "Documents en la cua d'impressio... "<< endl;
    queue<Document,list<Document> > CuaTemp(CuaDocsImpressio);
        while (!CuaTemp.empty()){
          Document D=CuaTemp.front();
        cout << D.getId() << "\t"<< D.getTitol() << "\t"<< D.getUsuari()
          <<endl;
           CuaTemp.pop();
        }
    }
}
```

Exemple: Programa per simular la cua d'una impressora (cont.)

```
    cout << "Sortir (S/N)?";
    char resposta; cin >> resposta;
    sortir=(resposta=='S'||resposta=='s');
  }
  return 0;
}
```

Edicions UPC
Inici
Contingut
Pàgina 312
Tornar
Pantalla Completa
Tancar
Sortir

Resum

- Cua és una estructura de dades bàsica i de molta utilitat en programació
- Prové de models reals ("cua de processos, persones, etc.") cosa que fa que tingui una interfície pròpia
- La peculiaritat de la cua és que es té accés només als extrems de la seqüència, al primer i últim element

  Aquesta peculiaritat fa que la cua es conegui com l'estructura FIFO (First In First Out), és a dir, primer en entrar és el primer a sortir
- La cua és d'aplicació quan s'ha de "donar servei" segons l'ordre d'arribada, com ara tractament de sol·licituds en una centre administratiu, trucades telefòniques, impressió de documents, etc.
- Tot i ser estructura de dades linial, en l'STL és un *adaptador* de contenidors seqüencials dequeu o list

Edicions UPC
Inici
Contingut
Pàgina 313
Tornar
Pantalla Completa
Tancar
Sortir

Criteris pràctics. Evitar errors

- Per tal de consultar els elements de la cua (per exemple, per fer un llistat de documents d'impressió) cal aplicar iteradament les operacions de la cua en aquest ordre: `front()` (per consultar) i `pop()` (per treure l'element)
- En aplicar `front()` i `pop()` es buida la cua per la qual cosa, per no perdre els elements, cal fer abans una còpia de la cua a través del constructor de còpia
- Abans d'aplicar les operacions de consulta i esborrat cal comprovar (usant el mètode `empty()`) que la cua no estigui buida
- D'altra banda, pel que fa les insercions, no hi ha límit, tret de la capacitat física de la memòria, ja que la cua en l'STL és una estructura de dades dinàmica
- A vegades podem necessitar usar més d'una cua en nostres programes, com ara, per simular el procés de projecció de pel·licules per una cadena, suposant que tenim la cua de la programació de pel·licules i volem aplicar el criteri que un cop emesa la pel·licula, no es pugui emetre fins que s'hagin emès totes les programades

Edicions UPC
Inici
Contingut
Pàgina 314
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítol 10,Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

15. Estructures de Dades: Contenidors associatius. Contenidor Map de l'STL

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - contenidors associatius, conceptes bàsics
  - classificació dels contenidors associatius
  - contenidor `map` de l'STL
  - operacions bàsiques de `map`
  - exemples d'ús
- **Resum**
- **Criteris pràctics. Evitar errors**
- **Referències**

Motivació. Relació amb el tema anterior

- En els temes anteriors hem vist contenidors seqüencials on cada element ocupa una posició en la seqüència

  La posició que ocupa un element no depèn del seu valor sinó del moment i lloc d'inserció a la seqüència

- Els contenidors associatius són una altra família de contenidors en què la posició d'un element depen del seu valor, segons un cert criteri d'ordenació

  L'ordre en què s'insereixen no importa!

- A causa de l'*ordenació automàtica*, són força apropiats quan necessitem fer cerques d'elements

- Contenidors "dissenyats" per recuperar la informació eficientment. A diferència dels contenidors seqüencials, no hi ha el mecanisme per inserir elements a una posició arbitrària

- L'STL ofereix quatre contenidors associatius:

  - `map`, `multimap`
  - `set`, `multiset`

- Veurem el `map`

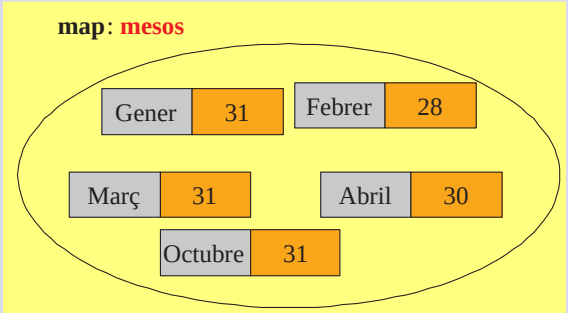
Conceptes bàsics contenidors associatius

- Els contenidors associatius gestionen les dades mitjançant *claus* que les identifiquen (p.ex. DNI identifica dades d'una persona) i permeten recuperació eficient de les dades usant les claus
- Els objectes s'emmagatzemen de manera ordenada en base als valors o les claus; els elements s'ordenen automàticament segons un criteri d'ordenació (per ex.: $<$ o una funció de comparació)
- L'STL proporciona quatre classes bàsiques de contenidors associatius depenent del tipus d'element que es gestiona i dels duplicats dels elements:
  - `set<T>`: una col·lecció en què els elements s'ordenen pel seu valor i no es permeten elements duplicats
  - `multiset<T>`: com `set` però suporta elements duplicats

- `map<clau, T>`: una col·lecció d'elements de forma <clau,valor>. Cada element té una clau que s'usa per l'ordenació. Suporta claus úniques. Es poden usar com *taules associatives*, taules indexades sobre elements d'un tipus qualsevol
- `multimap<clau, T>`: com `map`, però poden haver-hi claus duplicades (elements diferents que tenen la mateixa clau). Es poden usar com *diccionaris*, taules indexades sobre elements d'un tipus qualsevol

Exemples de map

- Un llistat de parells d'elements `<DNI,Persona>`
- Un magatzem de components electrònics manté parelles `<Ref,Component>`.
- La informació dels noms dels mesos i els dies que tenen (vegeu dibuix pàgina següent)



Operacions de map

- Comprovar si una clau està en l'estructura map
- Obtenir el valor associat a una clau
- Inserir un nou parell `<clau,valor>` a l'estructura de datos
- Eliminar una clau donada i el seu valor associat
- Saber quants parells `<clau,valor>` hi ha
- Saber si està buit o no

Edicions UPC
Inici
Contingut
Pàgina 320
Tornar
Pantalla Completa
Tancar
Sortir

Map en l'STL: `map<clau,T>`

- Cal indicar un tipus `clau` per a les claus i un tipus `T` per al valor associat amb la clau
- *Condició* sobre els valors de tipus `clau`: han de ser comparables
- D'altres condicions per als tipus `clau` i `T`:
  - constructor per defecte
  - constructor de còpia
  - operador d'assignació
- *Constructors* (que utilitzarem):
  - `map<tipus_clau,tipus_dades,[func_comparació]> nom_map;` Construeix un map buit. Per exemple, `map<string,int> mesos;`
  - les claus en el map estan ordenats segons `func_comparació`
  - pel cas de claus de tipus bàsics (`int`, `double`, ...) i els que tenen definit l'operador < com ara `string` no cal indicar-ho
  - constructor de còpia funciona com usualment

Operacions

Accés als elements

- A través de l'operador `[]`, que permet accedir als elements com si fos un array (map és un array associatiu)

  `T& M [Clau k]` retorna la referència del valor de tipus `T` associat a la clau `k`

  Per exemple:

  ```
  cout << "Abril te:  " << mesos["Abril"] << " dies";
  ```

  donaria el nombre de dies del mes d'abril

- A través d'iterador (bidireccional), de la manera usual

Nota: la clau dels elements del map és considerat constant, és del tipus `pair<const tClau,T>`
Motiu: l'ordenació automàtica. Això afecta la inserció, esborrat i modificació

Inserció

- `iterator M.insert(iterator pos, pair< Clau,T> elem)`

  Insereix `elem` al map `M` en cas que no estigui. L'iterador retornat apunta a l'element afegit

- `pair<iterator, bool> M.insert(pair< Clau,T> elem)`

  Insereix `elem` al map `M` en cas que no estigui.

  Retorna un parell (`pair`), on el primer element es l'iterador que apunta a l'element inserit o a l'existent en `M` i el segon diu si s'ha afegit (`true`) o ja existia (`false`)

Esborrat

- `void M.erase(iterator pos)` esborra l'element apuntat per l'iterador `pos`
- `void M.clear()` buida el map

Edicions UPC
Inici
Contingut
Pàgina 323
Tornar
Pantalla Completa
Tancar
Sortir

Modificació de la clau

Podem modificar la clau de l'element `<K,val>` de la següent manera:

- treure l'element amb la clau `K`
- inserir un nou element `<K',val>` on `K'` és el nou valor de la `clau` i `val` és el valor d'abans

Altres operacions

- `iterator M.find(Clau k)` retorna l'iterador que apunta a l'element `k` o `end()` si `k` no hi és a `M`
- `bool M.empty()`
- `int M.size()`

Exemple: Stock de llibres

```
#include<...>
#include<map>
//definim un tipus per les entrades a l'stock <titol,quantitat>
typedef pair<string,int> entrada;
int main(){
   map<string,int> stockLlibre; //stock de llibres
   entrada e1("El Quixot",100000); stockLlibre.insert(e1);
   entrada e2("C++ Senzill",200); stockLlibre.insert(e2);
   cout << "Quants exemplars hi ha a l'stock per \"El Quixot\"? "
                    << stockLlibre["El Quixot"] << endl;
   cout << "Quants exemplars hi ha a l'stock per \"C++ Senzill\"? "
                    << stockLlibre["C++ Senzill"] << endl;
   //Fem un llistat de l'stock
   map<string,int>::iterator iter = stockLlibre.begin();
   while (iter != stockLlibre.end()) {
      cout << "Llibre: "<< iter->first << " te en stock: "
                    << iter->second << " exemplars"<< endl;
      iter++;
   }
```

Exemple: Stock de llibres (cont.)

```
    //fem cerca pel títol de llibre
    map<string,int>::iterator iter2 = stockLlibre.find("El Quixot");
    if (iter2!= stockLlibre.end())
      cout << iter2 ->second ;
     return 0;
}
```

Map com array associatiu

- Els contenidors associatius no proveeixen accés directe; map fa una excepció!
- L'accés directe es proporciona usant la clau com a índex
- Hi ha dues diferències bàsiques:
  - a diferencia de les taules ordinàries, la clau com a índex pot ser de qualsevol tipus
  - l'índex no pot tenir un valor erroni (valor que no li correspon cap element a la taula); llavors, en cas de valor erroni, s'afegeix un element automàticament i s'inicialitza a través del constructor per defecte (pels tipus bàsics aquest valor és 0)

Edicions UPC
Inici
Contingut
Pàgina 326
Tornar
Pantalla Completa
Tancar
Sortir

Criteris de selecció de contenidors

Per a la solució d'un problema podem usar més d'un tipus de contenidor. Cal llavors tenir un cert criteri que justifiqui l'elecció:

- *accés a les dades*: si és important un accés aleatori, `vector` i `dequeu` són preferibles
- *inserció/esborrat de les dades*: si s'han de fer moltes insercions/esborrats de les dades (no només als extrems), `list` és l'opció
  Si les dades s'han d'inserir en algun ordre concret tenim `stack` i `queue` o `list`
- *variació del tamany de l'estructura de dades*: si la grandària de l'estructura anirà canviant durant l'execució del programa, l'opció és `list`. Altrament, `vector` o `dequeu` seria l'opció
- *dades ordenades*: si necessitarem de dades ordenades `map` i `set` són les opcions
- *cerca d'elements*: si necessitarem cercar freqüentment elements a l'estructura `map` i `set` són les opcions

Resum

- En moltes aplicacions podem necessitar mantenir la informació ordenada i recuperar-la eficientment. Els contenidors associatius són la solució
- La posició que ocupa un element depèn del seu valor i del criteri d'ordenació (no del moment ni lloc d'inserció)
- L'STL ofereix quatre tipus de tals contenidors: map i multimap, set i multiset
- El map manté parells d'elements `<clau,valor>` amb claus úniques en base de les quals els elements s'insereixen ordenadament en el map

  El map es pot recórrer amb iteradors en tots dos sentits
- Conceptualment, el map es una array associatiu

  Podem accedir al elements usant l'operdor [] d'arrays normals, ara, però, l'index pot prendre qualsevol valor d'un rang de valors amb l'operador $<$ definit

Criteris pràctics. Evitar errors

- Una manera senzilla d'inserir elements en el map és usar el constructe `pair` per definir els elements `<clau,valor>`, crear-ne parells d'elements i inserir-los amb el mètode `insert`
- Els maps es poden recórrer a través d'iteradors. L'iterador d'un map apunta al parell <clau,valor> que té dos membres: `first` i `second`. `first` correspon a la clau i `second` al valor associat amb la clau
- L'estructura map fa les insercions i les cerques eficientment (totes dues operacions en temps logarítmic)
- Cal tenir present que a l'hora de cercar una clau al map, si aquest no existeix s'afegeix al map amb valor el per defecte del tipus. O sigui, a diferència de les arrays normals, no es donaria un error, donant lloc a la inserció d'elements accidentalment

Edicions UPC
Inici
Contingut
Pàgina 330
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítol 6, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

16. Estructures de Dades: Algorismes bàsics de l'STL

- **Motivació. Relació amb el tema anterior**
- **Desenvolupament dels continguts del tema**
  - algorismes en l'STL
  - genericitat dels algorismes de l'STL
  - classificació dels algorismes
  - exemples d'ús
- **Resum**
- **Referències**

Edicions UPC
Inici
Contingut
Pàgina 331
Tornar
Pantalla Completa
Tancar
Sortir

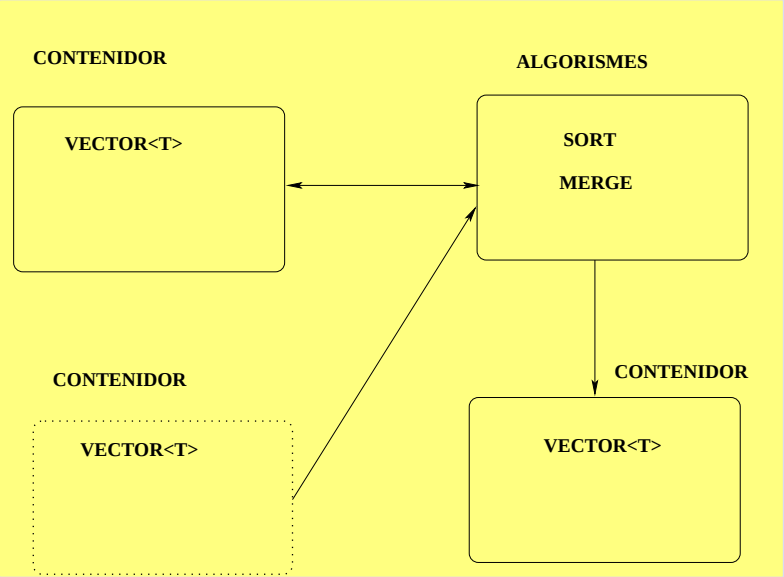
Motivació. Relació amb el tema anterior

- En els temes anteriors hem vist diferents tipus de contenidors (seqüencials i associatius)
- Una part important de l'STL són els algorismes que operen amb els contenidors
- Els algorismes ens permeten fer operacions de cerca, ordenació, modificació i recomptes, a més d'algorismes numèrics
- Els algorismes no formen part de cap contenidor concret (són funcions globals) i funcionen gràcies als iteradors que tenen els contenidors
- Cal saber quin algorisme funciona sobre quin contenidor (no val qualsevol combinació!). O, en cas de diferentes combinacions possibles, cal escollir la més eficient

Edicions UPC
Inici
Contingut
Pàgina 333
Tornar
Pantalla Completa
Tancar
Sortir

Algorismes i iteradors

Els algorismes de l'STL són genèrics, no estan lligats a cap contenidor en concret. Això és possible perquè els contenidors proporcionen iteradors per accedir als elements del contenidor



Classificació dels algorismes en l'STL

- Hi ha uns 60 algorismes oferts per l'STL
- Se solen agrupar segons diferents criteris: modificació o no del contenidor, modificació de l'ordre dels elements
- Les categories principals són:
  - algorismes no modificadors
  - algorismes modificadors
  - algorismes d'ordenació i cerca
  - algorismes numèrics
  - altres algorismes (auxiliars)
- Fitxers de capçalera pels algorismes de l'STL

  Per usar qualsevol algorisme, tret els numèrics, cal incloure la directiva `#include<algorithm>` (pels numèrics `#include<numeric>`)

Algorismes no modificadors

- Aquests algorismes no canvien ni l'ordre ni els elements del contenidor. Els més usats són:
  - `find(inici,final,valor)` – retorna la posició del primer element entre `inici` i `final` del valor indicat o `end` en cas de no trobar-se. Els contenidors associatius ja tenen el seu `find()`
  - `find_if(inici,final,cond)` – retorna la posició del primer element entre `inici` i `final` pel qual `cond(elem)=true` o `end` en cas de no trobar-se
  - `count(inici,final,valor)` –retorna el nombre d'elements del rang indicat que són iguals a `valor`
  - `count_if(inici,final,cond)` –retorna el nombre d'elements del rang indicat pel qual `cond(elem)=true`
  - `min_element(inici,final)`, – retorna la posició de l'element del valor mínim del rang indicat (usa l'operador <)
  - `min_element(inici,final,comp)`, – retorna la posició de l'element del valor mínim del rang indicat usant la funció de comparació `comp`
  - `max_element(inici,final)`, – retorna la posició de l'element del valor màxim del rang indicat (usa l'operador <)
  - `max_element(inici,final,comp)`, – retorna la posició de l'element del valor màxim del rang indicat usant la funció de comparació `comp`

Exemple d'ús

```
#include <...>
#include<list>
#include<algorithm>
bool MultipleDe3(int elem){
    return elem%3==0;
}
int main(){
     list<int> L;
     for (int i=1;i<=10;i++)
        L.push_back(i+10);
     list<int>::iterator pos;
     pos=find(L.begin(),L.end(),7);
     if (pos!=L.end())
       cout << "Element trobat: "<< *pos <<endl;
     else
       cout << "Element no trobat! "<<endl;
```

Exemple d'ús

```
    list<int>::iterator pos1;
    pos1=find_if(L.begin(),L.end(),MultipleDe3);
    if (pos1!=L.end())
      cout << "Element trobat: "<< *pos1 <<endl;
    else
      cout << "Element no trobat! "<<endl;
     cout << "Hi ha "<< count(L.begin(),L.end(),15) << " elements
     iguals a 15."<< endl;

    cout << "Hi ha "<< count_if(L.begin(),L.end(),MultipleDe3) <<
    " elements multiples de 3."<< endl;

     cout << "Element minim: "<< *min_element(L.begin(),L.end());
    cout << " Element maxim: "<< *max_element(L.begin(),L.end());
    return 0;
}
```

Edicions UPC
Inici
Contingut
Pàgina 337
Tornar
Pantalla Completa
Tancar
Sortir

Algorismes modificadors

- Aquests algorismes canvien els valors dels elements (d'un rang especificat) del contenidor o modifiquen el contenidor (operacions d'esborrat)
- Els més usats són:
  - `replace(inici,final,valorAntic,valorNou)`
  - `replace_if(inici,final,cond,valorNou)`, on `cond` és una condició que han de complir els elements
  - `remove(inici,final,valor)` –no canvia el nombre dels elements (fa un reemplaçament dels elements "removed" pels elements següents que no són "removed". `erase()`, en canvi, canvia el nombre dels elements
  - `remove_if(inici,final,cond)`
  - `unique(inici,final)`, treu duplicats consecutius. En cas d'estar el contenidor ordenat, treu tots els duplicats
  - `unique(inici,final,cond)`

Nota: els algoritmes de tipus `remove()` no són aplicables als contenidors associatius, els quals proporcionen `erase()`

Exemple d'ús

```
#include <...>
#include<string>
#include <list>
#include <algorithm>
void omplirLlista(list<string>&LS){
string s; cin>>s;
     while (s!="Fi"){
        LS.push_back(s);
        cin >> s;
     }}
void mostrarLlista(list<string>&LS) {
    list<string>::iterator LSI=LS.begin();
    while(LSI!=LS.end()){
       cout << *LSI << ' ';
         LSI++;
    }
    cout << endl;
    }
```

Exemple d'ús (cont.)

```
bool InicialT(string s){
    return s[0]=='T';
}
int main(){
    //Llista per noms de ciutats
    list <string> LS;
    omplirLlista(LS);
    cout << "Llista introduida: "<< endl;
    mostrarLlista(LS);
    cout << "Introdueix el valor a reemplacar: ";
    string s1; cin >> s1;
    cout << "Introdueix el valor nou: ";
    string s2; cin >> s2;
    replace(LS.begin(),LS.end(),s1,s2);
    cout << "Llista despres de replace: "<< endl;
    mostrarLlista(LS);
```

Exemple d'ús (cont.)

```
    cout << "Introdueix el valor nou: ";
    string s; cin >> s;
    replace_if(LS.begin(),LS.end(),InicialT,s);
    cout << "Llista despres de replace_if: "<< endl;
    mostrarLlista(LS);
    cout << "Introdueix el valor a eleminar: ";
    string st; cin >> st;
    remove(LS.begin(),LS.end(),st);
    cout << "Llista despres de remove: "<< endl;
    mostrarLlista(LS);
    return 0;
}
```

Algorismes d'ordenació i cerca

Aquests algorismes canvien l'ordre dels elements (d'un rang especificat) del contenidor. Els més usats són:

- `sort(inici,final)` –ordena els elements del rang especificat segons l'ordre `<`
- `sort(inici,final,comp)` –ordena els elements del rang especificat segons l'ordre `comp`
- `binary_search(inici,final,valor)` –retorna si existeix l'element igual a `valor`
- `binary_search(inici,final,cond)` –retorna si existeix l'element que compleix `cond`
- `merge(inici1,final1,inici2,final2,inici)` –mescla els dos contenidors ordenats en un contenidor ordenat (segons `<`)
- `merge(inici1,final1,inici2,final2,inici,comp)` –mescla els dos contenidors ordenats en un contenidor ordenat segons el criteri d'ordenació `comp`

Edicions UPC
Inici
Contingut
Pàgina 343
Tornar
Pantalla Completa
Tancar
Sortir

Observacions:

- Hi ha les versions `stable_sort` que garanteixen que l'ordre dels elements iguals no és alterat i també `partial_sort` per fer una ordenació parcial
- Important: `sort` és aplicable als contenidors que tenen un iterador d'accés aleatori (`vector`, `deque`). El contenidor `list` ja ofereix el seu `sort()`
- `list` ofereix el seu `merge`

Exemple d'ús

```
int main(){
    vector <string> VS;
    omplir(VS);
    cout << "Llista introduida: "<< endl;
    mostrar(VS);
    sort(VS.begin(),VS.end());
    cout << "Llista despres d'ordenar: "<< endl;
    mostrar(VS);
    //La llista està ordenada. Fem cerca binària
    cout << "Introduiex l'element a buscar: ";
    string str; cin>> str;
    if(binary_search(VS.begin(),VS.end(),str))
      cout << str << " esta en la llista";
    else
       cout << str << " no esta en la llista"<<endl;
```

Exemple d'ús (cont.)

```
    //Declarem un nou vector per fer prova del merge
    vector <string> VS1;
    //Omplim el vector
    omplir(VS1);
    //Ordenem el vector
    sort(VS1.begin(),VS1.end());
    //Resultat de la mescla
    vector <string> VS2(VS.size()+VS1.size());
    //Apliquem merge
    merge(VS.begin(),VS.end(),VS1.begin(),VS1.end(),
    VS2.begin());
    cout << "Despres del merge: "<< endl;
    mostrar(VS2);
    return 0;
}
```

Algorismes numèrics

Els algorismes numèrics fan (bàsicament) processaments numèrics de contenidors amb elements de valors numèrics Els bàsics són:

- `acummulate(inici,final,valInicial)` –retorna la suma de `valInicial` amb els elements entre `inici` i `final`. Així, per `valInicial=0` ens donaria la suma dels elements del contenidor
- `acummulate(inici,final,valInicial,op)` –retorna el resultat de `valInicial op elem1 op elem2...` amb els elements entre `inici` i `final`. Per `op=multiplicar` i `valInici=1` donaria el producte dels elements
- `inner_product(inici1,final1,inici2,valInicial)` –retorna el producte interior ($\sum a_i * b_i$) dels elements + el `valInicial`. Així, per dos contenidors vectors i `valInicial=0` donaria el producte escalar dels vectors
- `partial_sum(inici,final,iniciNou)` –calcula les sumes parcials ($a1, a1 + a2, a1 + a2 + a3, \ldots$) per cadascún dels elements entre `inici` i `final` i deixa el resultat a partir de l'`iniciNou`

Exemple d'ús

```
#include<...>
#include <vector>
#include <iterator>
#include <numeric>
//Nota: es pot usar multiplies<int>()
int Multiplica(int x,int y){
return x*y;
}
typedef ostream_iterator < int> IntOstreamIt;
int main(){
    //ostream iterator
    IntOstreamIt itOstream(cout," ");
    vector <int> VI;
    omplir(VI);
    cout << "Elements introduits: "<< endl;
    mostrar(VI);

    cout << "Suma dels elements del vetor: "<<
    accumulate(VI.begin(),VI.end(),0) << endl;
    cout << "Producte dels elements del vetor: "<<
    accumulate(VI.begin(),VI.end(),1,Multiplica) << endl;
```

Exemple d'ús (cont.)

```
    vector <int> VI2;
    omplir(VI2);

    cout << "Producte escalar: "<<
    inner_product(VI.begin(),VI.end(),VI2.begin(),0) << endl;
   //Imprimeix les sumes parcials de VI
   partial_sum(VI.begin(),VI.end(),itOstream);

   //Sumes parcials guardades en un vector
    vector<int> VSP(VI.size());
    partial_sum(VI.begin(),VI.end(),VSP.begin());
    mostrar(VSP);
    return 0;
}
```

Resum

- Un dels tres components de la lliberia STL és el dels algorismes

  L'STL proporciona una varietat d'algorismes genèrics, és a dir, que són d'aplicació sobre els contenidors amb independència del tipus dels objectes que emmagatzema el contenidor

  Així, per exemple, l'algorisme `sort()` s'ha implementat una sola vegada i es pot aplicar sobre qualsevol contenidor el tipus dels elements del qual té implementat l'operador $<$

- Hi ha vàries famílies d'algorismes, tots ells englobats en la llibreria `algorithm` (els algorismes numèrics es troben en la llibreria `numerics`)
- La implementació dels algorismes genèrics es fa possible gràcies al mecanisme dels iteradors
- Per tal d'aplicar els algorismes de l'STL, el tipus dels elements ha de proporcionar els operadors necessaris com ara el ==, els de comparació, etc.

Referències

- *Bàsiques*:
  - L. Joyanes. *Programación en C++*. Capítol 25, McGraw-Hill, 2002.
- *Complementàries*:
  - Josuttis, Nicolai M. *The C++ standard library a tutorial and handbook*. Capítol 6, Reading, Massachusetts Addison-Wesley, 1999 (disponible biblioteca BRGF, BCN, versió paper i digital).

17. Cas d'estudi: Programa per realitzar operacions bàsiques amb una imatge. Canvis –fer/desfer– recents d'una imatge

- **Motivació. Relació amb els temes anteriors**
- Desenvolupament del cas d'estudi
  - repàs de conceptes bàsics
  - especificació de les classes necessàries
  - implementació de les classes:
    - llibreria **WINBGIM** (pel compilador gratuït mingw32 GNU C++). http://www.cs.colorado.edu/∼main/bgi/
    - llibreria SDL (**Simple DirectMedia Layer**, http://www.libsdl.org/)
- **Resum**
- **Referències**

Motivació

- En els temes anteriors hem vist diferents estructures de dades (contenidors seqüencial i un d'associatiu)
- Es tracta de desenvolupar un programa per realitzar operaciones senzilles amb una imatge i poder fer/desfer els canvis recents aplicats a la imatge
- L'objectiu principal és el d'aplicar a la pràctica els conceptes vistos fins ara per resoldre un problema concret:
  - identificar, especificar i implementar classes
  - escollir estructures de dades adequades
  - promoure l'ús de llibreries (de codi obert)

Nota: L'objectiu del cas d'estudi és fer participar els estudiants en la solució d'un problema que requereix els conceptes apresos al llarg de l'assignatura. També s'aprofitaria el cas d'estudi per fer èmfasi en les fases del desenvolupament del programa, cerca de la informació necessària i la documentació del programa. Per això només es donen les pautes bàsiques de desenvolupament d'un cas d'estudi el qual seria completat al llarg de 3h

L'enunciat del problema

Es tracta d'implementar un programa en C++ per a operar amb imatges. En executar-se, el programa ha de mostrar un menú contextual:

1. Recuperar una imatge en format ppm del disc
2. Assignar una descripció (títol, autor, data de creació) a la imatge.
3. Consultar la descripció de la imatge
4. Modificar la descripció de la imatge
5. Consultar les dimensions de la imatge
6. Fer display de la imatge a la pantalla
7. Modificar un píxel de la imatge i tornar a dibuixar-la
8. Desplaçar la imatge horizontalment
9. Desplaçar la imatge verticalment
10. Treure un requadre de la imatge
11. Desfer les últimes k accions fetes amb la imatge, essent les accions: modificació d'un pixel, desplaçament horizontal, desplaçament vertical
12. Sortir del programa

L'enunciat del problema (cont.)

Per a implementar el programa disposarem d'un mòdul definit que proporciona les següents primitives:

- llegir una imatge en format ppm des del disc i guardar-la en una matriu de píxels essent el píxel en la representació RGB
- escriure (guardar) una imatge representada per un matriu de píxels a imatge en format ppm
- fer display d'una imatge representada per una matriu de pixels

Organització del cas d'estudi

- *Part I*: repàs d'alguns conceptes bàsics relacionats
- *Part II*:
  - identificació de les classes
  - especificació de les classes
- *Part III*: implementació de les classes i del programa principal
  - amb la llibreria **WINBGIM** (pel compilador gratuït **mingw32 GNU C++**)
  - alterntivament, amb les primitives de la llibrera SDL (**Simple DirectMedia Layer**)

Cas d'estudi: Part I

- Repàs de conceptes bàsics:
  - colors primaris
  - píxel
  - imatge
  - models de colors (RGB, HSV, YIQ, CMY)
  - transformacions entre els models de colors (RGB $\leftrightarrow$ HSV $\leftrightarrow$ YIQ $\leftrightarrow$ CMY)
  - alguns formats d'imatges

Cas d'estudi: Part II

- Identificació de les classes
  - Pixel, Imatge, Accio, Descripcio
- Especificació de les classes
- Especificació de la classe Pixel:
  - estructura de dades per representar un píxel
  - funcions per consultar els valors segons el model de color
  - funcions per assignar/consultar els valors segons el model de color
  - funcions per ajustar els valors
  - funcions per convertir un píxel d'un model a un altre
  - dibuixar un píxel a la pantalla

Cas d'estudi: Part II (cont.)

- *Especificació de la classe Imatge*:
  - consultar la descripció de la imatge (autor, data creació,...)
  - consultar l'amplada i alçada d'una imatge
  - consultar el pixel d'una posició
  - modificar un pixel d'una posició
  - desplaçar una imatge horitzontalment
  - desplaçar una imatge verticalment
  - dibuixar un recuadre de la imatge
  - mantenir les k últimes accions fetes amb la Imatge i permetre desfer-les
  - llegir una imatge des d'un arxiu
  - dibuixar una imatge a la pantalla

Edicions UPC
Inici
Contingut
Pàgina 359
Tornar
Pantalla Completa
Tancar
Sortir

Cas d'estudi: Part II (cont)

- *Especificació de la classe Accio*:
  - mantenir la informació d'una acció
  - consultar la informació d'una acció
- *Especificació de la classe Descripció*

Cas d'estudi: Part III

Implementació de les classes i del programa principal:

- amb la **WINBGIM Library** (pel compilador gratuït **mingw32 gnu C++**)
- amb les primitives de la llibreria SDL (**Simple DirectMedia Layer**)

Part I: Apunts sobre els conceptes bàsics

- *Colors*
  - la llum és descomposta en les intensitats que corresponen als colors *vermell*, *verd*, i *blau*
  - aquest trio de colors es denominen els colors primaris
  - a partir dels colors primaris podem aconseguir altres tres colors: *cian*, *groc* i *magenta*, anomenats colors complementaris
- *Píxels*
  - els colors són combinats per a formar tons i d'altres colors en una àrea petita
    Aquesta àrea és anomenada "píxel", que ve de l'anglès: *picture element* o element pictogràfic (o d'imatge)
  - el píxel és la part fonamental més petita que conté informació per a crear una imatge en un sistema gràfic

Apunts sobre els conceptes bàsics (cont.)

- *Imatges*
  - un conjunt de píxels forma una malla per a crear una imatge completa
  - la majoria dels sistemes gràfics usa una malla rectangular amb les mateixes dimensions d'una superfície rectangular: amplada i alçada
  - l'amplada és el nombre de columnes i l'alçada, el nombre de files de tal superfície gràfica
  - el nombre de píxels en l'amplada i en l'alçada constitueixen la resolució (gràfica) de la imatge
  - per exemple, 640 x 480 significa que la resolució gràfica de la imatge constitueix 640 columnes i 480 files de píxels formant una malla de 307.200 píxels, i, 800 x 600 significa que obtenim una imatge de 480.000 píxels (imatge de major resolució gràfica per tant, millor qualitat d'imatge)

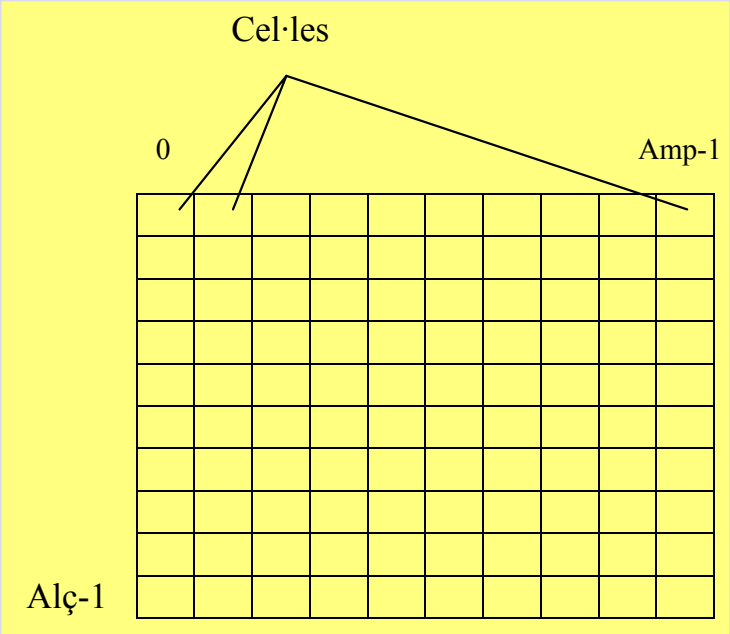
Exmple d'una imatge: *Mosaic de Gaudí*

Les "peces" petites són els píxels de la imatge

Edicions UPC
Inici
Contingut
Pàgina 363
Tornar
Pantalla Completa
Tancar
Sortir

Apunts sobre els conceptes bàsics (cont.)

Representació d'una imatge amb matriu bidimensional



Apunts sobre els conceptes bàsics (cont.)

- *Models de colors*
  - en sistemes gràfics, els colors són descompostos i representats per combinacions de valors numèrics
  - generalment, es parla d'un color diferent per a cada combinació de valors (en realitat, s'està parlant de tons diferents de colors)
  - per exemple, si es parla d'un sistema gràfic de 15 *bits per píxel* (bpp) tenim a la nostra disposició 32.768 colors, encara que en realitat siguin uns quants colors i diversos milers de tons
  - els models més populars i usats són *RGB*, *HSV*, *CMY* i *YIQ*

Apunts sobre els conceptes bàsics (cont.)

- *Model RGB* (Red, Green, Blue): aquest trio de colors intenta modelar el sistema visual humà. La majoria dels monitors es basen en aquest model, pel fet que estan basats en l'emissió de la llum
- *Model HSV* (Hue, Saturation, Intensity): en lloc d'usar un model tricolor per a formar altres colors i tons, aquest model es basa en tres propietats que serveixen per a definir els colors que percebem
- *Model CMY* (Cyan, Magenta, Yellow): són complementaris als RGB
- *Model YIQ* (Brightness, Chroma, Purity): aquest model es basa en el model RGB, però restringint i descomponent alguns valors d'RGB

Alguns formats d'imatges

PPM: *Portable Pix Map* file format. En aquest cas, per a cada cel· la s'emmagatzemen tres valors que corresponen a la quantitat de vermell, verd i blau que es barregen per a obtenir la imatge desitjada
BMP: *BitMapPixel*
PNG: *Portable Network Graphics*
JPEG: *Joint Photographic Experts Group*

Part II: Especificació de classes

```
//Fitxer ACCIO.HPP
//Especificació de la classe acció per gestionar una acció que es fa amb
//la imatge per després desfer-la
//Es  consideraran les següents accions: modificar píxel,
// desplaçament horizontal, desplaçament vertical
#ifndef ACCIO_H
#define ACCIO_H
#include<iostream>
//Definim el tipus d'acció: 0=modifPix, 1=desplH, 2=desplV
typedef enum tipusaccio{modifPix,desplH,desplV};
class accio{
      tipusaccio ta;    //Tipus acció
      int x,y;          //Coordenades del pixel, pel cas de l'acció=modifPix
      double R,G,B;     //Valors del pixel abans de ser modificat
      int dh;           //Deplaçament horizontal, pel cas de l'acció=desplH
      int dv;           //Deplaçament vertical, pel cas de l'acció=desplV
```

Edicions UPC
Inici
Contingut
Pàgina 368
Tornar
Pantalla Completa
Tancar
Sortir

```
public:
//constructors
//constructor per defecte
accio();
//Constructor per l'acció de modificar un pixel
//Rep les coordenades del pixel i els valors RGB
//Crea un objecte accio de tipus modifPix amb els
//valors d'x,y,R,G,B
accio(int x,int y,double R,double G,double B);
//Constructor per l'acció de deplaçament
//Rep el tipus de desplaçament i la quantitat de la mateixa
//Crea un objecte accio del tipus donat amb el valor de desplaçament
 accio(tipusaccio ta,int d);
 //Constructor de còpia (de fet no fa falta)
 accio(const accio& A);
//Destructor
~accio();
```

```
    //Mètodes consultors
    tipusaccio getTipusAccio() const;
    int getX() const;
    int getY() const;
    double getR() const;
    double getG() const;
    double getB() const;
    //Mètode per consultar el desplaçament horizontal
    int getDesplH() const;
    //Mètode per consultar el desplaçament vertical
    int getDesplV() const;
    //Operador d'assignació (no fa falta en aquest cas)
    //Operador de sortida
    friend ostream& operator <<(ostream& os,const accio& A);
};
#endif
```

Part II: Especificació de classes (cont.)

```
//Pixel.h
//Especificació de la classe Pixel
//Es consideren  diversos models de colors i la transformació entre ells
#ifndef PIXEL_H
#define PIXEL_H
//Definim diferents models de colors
// 0=RGB, 1=HSV, 2=YIQ, 3=CMY
typedef enum {RGB, HSV, YIQ, CMY} ModelColor;
class Pixel{
   //Model del pixel
   ModelColor model;
   //RGB
   double red,green,blue;
   //HSV
   double  hue,saturation,intensity;
   //YIQ
   double  brightness,chroma, purity;
```

Part II: Especificació de classes (cont.)

```
  //CMY
  double cyan, magenta, yellow;
  //Mètode privat per ajustar els valors entre 0.0 i 1.0
   void ajustarValors(double& valor1,double& valor2,double& valor3);
public:
  //Construtor per defecte
  Pixel ();
  //Constructor de còpia (no fa falta)
  // Destructor
  ~Pixel ();
  // Mètodes consultors
  //Mètode per consultar el model del píxel
  ModelColor getModel();
  // Model RGB
  double getRed ();
  double getGreen ();
  double getBlue ();
```

Part II: Especificació de classes (cont.)

```
// Model HSV
double getHue ();
double getSaturation ();
double getIntensity ();
// Model YIQ
double getBrightness ();
double getChroma ();
double getPurity ();
// Model CMY
double getCyan ();
double getMagenta ();
double getYellow ();
// Mètodes modificadors
void setRGB (double red, double green, double blue);
void setHSV (double hue, double saturation, double intensity);
void setYIQ (double brightness, double chroma, double purity);
void setCMY (double cyan, double magenta, double yellow);
```

Part II: Especificació de classes (cont.)

```
// Mètodes modificadors per ajustar els valors un a un
 // Model RGB
  void ajustarRed   (double valor);
  void ajustarGreen (double valor);
  void ajustarBlue  (double valor);
  // Model HSV
  void ajustarHue        (double valor);
  void ajustarSaturation (double valor);
  void ajustarIntensity  (double valor);
  // Model YIQ
  void ajustarBrightness (double valor);
  void ajustarChroma     (double valor);
  void ajustarPurity     (double valor);
  // Model CMY
  void ajustarCyan    (double valor);
  void ajustarMagenta (double valor);
  void ajustarYellow  (double valor);
```

Part II: Especificació de classes (cont.)

```
    //Mètode per passar/convertir d'un model a l'altre
    void convertirA(ModelColor nouModel);
    //Operador d'assignació (no fa falta)
};
#endif
```

Part II: Especificació de classes (cont.)

- *Classe Descripcio*
  - títol de la imatge
  - nom autor
  - cognom autor
  - nacionalitat
  - data creació
- *Classe Imatge*

Classe Descripcio

```
//Classe Descripcio.hpp
//Gestiona informació associada a una imatge
#ifndef DESCRIPCIO_HPP
#define DESCRIPCIO_HPP
#include<iostream.h>
class Descripcio{
      char* titol;  //títol de la imatge
      char* nom;   //nom autor
      char* cognom;  //cognom autor
      char* nacionalitat; //nacionalitat autor
      //Data creació
      int dia;
      int mes;
      int any;
  public:
     //Constructor per defecte
     Descripcio();
     //Constructor amb paràmetres
     Descripcio(char* T, char* N, char* C, char* Na,int d, int m, int a);
     //Constructor de còpia
     Descripcio(const Descripcio& D);
```

Classe Descripcio (cont.)

```
    //Destructor
    ~Descripcio();
    //Mètodes consultors
    char* getTitol();
    char* getNom();
    char* getCognom();
    char* getNacionalitat();
    //Consultar data
    int getDia();
    int getMes();
    int getAny();
    //Mètodes modificadors
    void setTitol(char* T);
    void setNom(char* N);
    void setCognom(char* C);
    void setNacionalitat(char* Na);
    //Operador d'assignació
    Descripcio& operator = (const Descripcio&D);
    //Operador de sortida
    friend ostream& operator <<(ostream& os,const Descripcio& D);
};
#endif
```

Classe Descripcio (cont.)

```
//Classe Descripcio.cpp
//Gestiona informació associada a una imatge
#include"Descripcio.hpp"
//Constructor per defecte
Descripcio::Descripcio(){}
//Constructor amb paràmetres
Descripcio::Descripcio(char* T, char* N, char* C, char* Na,int d, int m, int
a){
                titol=new char[strlen(T)+1];
                strcpy(titol,T);
                nom=new char[strlen(N)+1];
                strcpy(nom,N);
                cognom=new char[strlen(C)+1];
                strcpy(cognom,C);
                nacionalitat=new char[strlen(Na)+1];
                strcpy(nacionalitat,Na);
                dia=d;mes=m;any=a;
  }
```

Edicions UPC
Inici
Contingut
Pàgina 378
Tornar
Pantalla Completa
Tancar
Sortir

Classe Descripcio (cont.)

```
//Constructor de còpia
Descripcio::Descripcio(const Descripcio& D){
        titol=new char[strlen(D.titol)+1];
        strcpy(titol,D.titol);
        nom=new char[strlen(D.nom)+1];
        strcpy(nom,D.nom);
        cognom=new char[strlen(D.cognom)+1];
        strcpy(cognom,D.cognom);
        nacionalitat=new char[strlen(D.nacionalitat)+1];
        strcpy(nacionalitat,D.nacionalitat);
        dia=D.dia;mes=D.mes;any=D.any;
}
//Destructor, alliberem espai
Descripcio::~Descripcio(){
        delete[] titol;
        delete[] nom;
        delete[] cognom;
        delete[] nacionalitat;
}
```

Classe Descripcio (cont.)

```
//Mètodes consultors
char* Descripcio::getTitol(){return titol;}
char* Descripcio::getNom(){return nom;}
char* Descripcio::getCognom(){return cognom;}
char* Descripcio::getNacionalitat(){return nacionalitat;}
//Consultar data
int Descripcio::getDia(){return dia;}
int Descripcio::getMes(){return mes;}
int Descripcio::getAny(){return any;}
//Mètodes modificadors
void Descripcio::setTitol(char* T){
        delete[] titol;
        titol=new char[strlen(T)+1];
        strcpy(titol,T);
}
void Descripcio::setNom(char* N){
        delete[] nom;
        nom=new char[strlen(N)+1];
        strcpy(nom,N);
}
```

Classe Descripcio (cont.)

```
void Descripcio::setCognom(char* C){
        delete[] cognom;
        cognom=new char[strlen(C)+1];
        strcpy(cognom,C);
}
void Descripcio::setNacionalitat(char* Na){
        delete[] nacionalitat;
        nacionalitat=new char[strlen(Na)+1];
        strcpy(titol,Na);
}
```

Edicions UPC
Inici
Contingut
Pàgina 381
Tornar
Pantalla Completa
Tancar
Sortir

Classe Descripcio (cont.)

```
//Operador d'assignació
Descripcio& Descripcio::operator = (const Descripcio&D){
        if(this!=&D){
            delete[] titol;
            titol=new char[strlen(D.titol)+1];
            strcpy(titol,D.titol);
            delete[] nom;
            nom=new char[strlen(D.nom)+1];
            strcpy(nom,D.nom);
            delete[] cognom;
            cognom=new char[strlen(D.cognom)+1];
            strcpy(cognom,D.cognom);
            delete[] nacionalitat;
            nacionalitat=new char[strlen(D.nacionalitat)+1];
            strcpy(nacionalitat,D.nacionalitat);
            dia=D.dia;mes=D.mes;any=D.any;
        }
        return *this;
}
```

Edicions UPC
Inici
Contingut
Pàgina 382
Tornar
Pantalla Completa
Tancar
Sortir

Classe Descripcio (cont.)

```
//Operador de sortida
ostream& operator <<(ostream& os,const Descripcio& D){
        os << D.titol << ' '<< D.nom << ' '<< D.cognom << ' '<<
        D.nacionalitat << endl;
        os << D.dia << '/'<< D.mes << '/'<< D.any;
        return os;
}
```

Edicions UPC
Inici
Contingut
Pàgina 383
Tornar
Pantalla Completa
Tancar
Sortir

Part III: Implementació

- *Implementació de les classes*
- *Implementació del programa principal*

Edicions UPC
Inici
Contingut
Pàgina 385
Tornar
Pantalla Completa
Tancar
Sortir

Referències

- The WINBGIM Library for the free mingw32 gnu C++ compiler
  http://www.cs.colorado.edu/~main/bgi/
- Simple DirectMedia Layer
  http://www.libsdl.org/