PUBLICACIÓ DOCENT

MANUAL DE LABORATORI D'ESIN Sessió 4

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

Vilanova i la Geltrú, 1 d'octubre de 2019

4

Exercici

L'objectiu d'aquest exercici és resoldre problemes usant llistes amb iteradors implementades per nosaltres mateixos usant memòria dinàmica. És un exercici curt perquè tingueu temps de repassar i acabar les sessions anteriors.

Caldrà resoldre el següent problema de la plataforma jutge.org; el trobareu en l'apartat de Piles, Cues i Llistes del curs ESIN (Vilanova):

- Capicues (https://jutge.org/problems/P77686_ca)

Els problemes d'aquesta secció no poden utilitzar la classe `list` de la STL. Cal incloure la definició i implementació pròpia de la classe llista amb iteradors genèrica amb memòria dinàmica (la trobareu en els apunts de teoria; per evitar problemes copiant des de fitxers PDF les podeu copiar de la carpeta `/home/public/esin/sessio4`). A més a més de l'especificació i implementació de la classe `llista_itr` (fitxers `llista_itr.hpp` i `llista_itr.t`), cal implementar la funció `bool es_capicua(int n)` i el programa principal que la crida (`main.cpp`); aquesta funció afegirà els elements en una llista per després comparar-los si són capicua usant un parell d'iteradors (per comparar el primer amb l'últim, el segon amb el penúltim, ...).

Degut a que jutge.org només permet l'enviament d'un fitxer amb la solució del problema, en el mateix fitxer hi ha d'haver l'especificació i la implementació de la classe i la funció `bool es_capicua(int n)`. I també cal eliminar les directives `#include "llista_itr.t"` i `#include "llista_itr.hpp"` per no tenir problemes de precompilació. Ho pots fer tot a la vegada amb la comanda:

```
cat llista_itr.hpp llista_itr.t main.cpp | sed '/include "llista_itr./d' > solucio.cpp
```

i enviar a jutge.org el fitxer `solucio.cpp`.

Com que possiblement aquest problema ja el teniu resolt des de les pràctiques de PRO1, envieu la nova solució a jutge.org amb l'anotació "Fet amb la classe llista_itr memòria dinàmica" perquè el professor sàpiga quina versió mirar quan us la corregeixi.

D

Estil de programació i documentació

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

D.1 Noms de variables adequats

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

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

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

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

[1] "Indentació".

```
    }
}
```

és un codi molt menys entenedor que

```
void inicia_taula(elem a[], int n) {
   for (int i = 0; i < n; i++) {
      a[i] = i;
   }
}
```

D.2 Conveni consistent pels identificadors

"Inventeu" un esquema per expressar els identificadors i apliqueu-lo sistemàticament i consistent.

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

```
template <typename T>
class pila {
   ...
  private:
      int _cim;
      T _cont;
      int _max_elems;
};
```

D.3 Utilitzar noms en forma activa per les funcions

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

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

```
conjunt c;
if (c.pertany(x))
...
```

amb

```
conjunt c;
if (c.conte(x))
...
```

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

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

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

```
class Pila {
 public:
   Pila(int max_elements);
   ~Pila();
   Pila(const Pila& s);
   Pila& operator=(const Pila& la_pila);

   void Apilar(int element);
   int desapila();
   int Cim();
   bool Buida();

 private:
   struct tnodo {
     int info;
     tnode* seguent;
   };
   tnode* cim;
   int Nr_Elems_Pila;
   int maxElems;
};
```

D.4 Nom d'identificadors precisos

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

Per exemple, la implementació següent no és gens adequada:

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

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

D.5 Identació del codi

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

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

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

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

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

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

```
if (mes == FEBRER) {
   correcte = true;
   if (es_any_traspas(any))
      if (dia > 29)
         correcte = false;
   else     // aquest else NO emparella amb el segon if!!
      if (dia > 28)
         correcte = false;
}
```

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

```
if (mes == FEBRER) {
   correcte = (dia <= 28) or
              (dia == 29 and es_any_traspas(any));
}
```

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

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

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

```
    {
        ...
    }
}
```

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

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

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

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

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

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

```
if (B_1)
    S_1
else if (B_2)
    S_2
...
else if (B_n)
    S_n
else
    S_{n+1}
```

amb tots els `else`'s alineats; quan les B_i's són mútuament excloents (els `if` ... `else`'s s'avaluen seqüencialment i no hi ha indeterminisme).

D.6 Evitar una lògica del programa antinatural

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

Per exemple, en lloc de

```
int s = 0;
node* p = _primer;
if (p == NULL) {   // la llista és buida; no fem res
}
else {
    while (p != NULL) {
        s = s + p -> valor;
        p = p -> seg;
    }
}
return s;
```

hauríem d'haver escrit:

```
int s = 0;
node* p = _primer;
while (p != NULL) {
    s += p -> valor;
    p = p -> seg;
}
return s;
```

o bé

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

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

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

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

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

D.7 Disminuir la complexitat

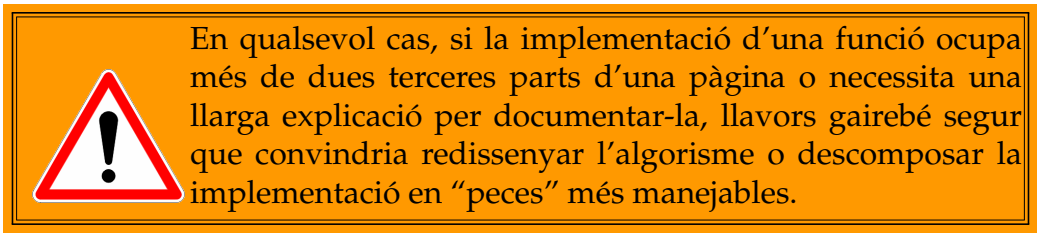
Disminuïu la complexitat del vostre codi mitjançant un ús assenyat de la descomposició funcional. Aprofiteu les solucions a problemes similars mitjançant una adequada descomposició funcional.

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

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

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

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



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

D.8 Useu construccions similars per tasques similars

Useu construccions similars per realitzar tasques similars.

Si en un punt del programa inicieu una taula *t* mitjançant:

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

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

```
i = 0;
nnuls = 0;
while (i <= n - 1) {
    if (A[i] != 0) {
        ++nnuls;
    }
    ++i;
}
```

D.9 No usar variables globals

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

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

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

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

```
// variables globals
int t[20];
int n;

// retorna cert si i només si 'x' està en t[0..n-1]
bool esta(int x) {
    for (int i = 0; i < n and t[i] != x; ++i)
        ;          // el cos del bucle és buit
    return i < n;
}
```

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

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

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

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

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

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

En particular podem usar variables privades de classe :

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

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

- no poden ser inicialitzades en un únic pas, o

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

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

D.10 Utilitzar variables locals

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

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

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

```
27     return conte_rec(x, p -> sig);
28 }
```

D.11 Codi ben estructurat

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

Tampoc és un bon estil "trencar" la iteració modificant la variable de control que recorre la seqüència:

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

Les úniques desviacions acceptables respecte a aquesta regla són:

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

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

es pot escriure

```
if (i > 1) {
   return x;
}
else if (i == 1)  {
   return y;
}
else {   // if (i ¡ 1)
   return z;
}
```

o fins i tot

```
if (i > 1) {
   return x;
}
if (i == 1) {
   return y;
}
return z;
```

D.12 Bona documentació

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

Per exemple,

```
class llista {
      ...
  private:
      struct node {
         string clau;
         int valor;
         node* seg;
      };
      ...
      nodo* _primer;
};
```

no ens diu si la llista està implementada mitjançant una llista simplement enllaçada, si està tancada circularment o no, si hi ha o no un node fantasma, si està ordenada o no

creixentment pel camp `clau` de cada node, etc. Per tant, s'ha de documentar adequadament la forma en que la representació serà utilitzada.

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

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

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