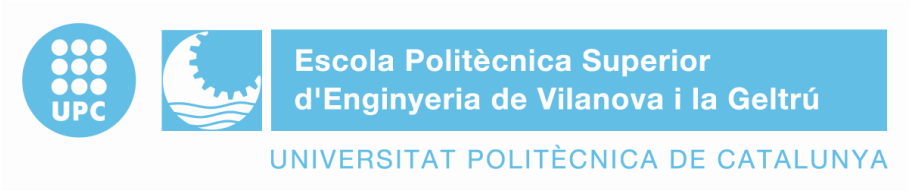

PUBLICACIÓ DOCENT

MANUAL DE LABORATORI D'ESIN Sessió 2

AUTOR: Bernardino Casas, Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Q3
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
ANY: 2020

Vilanova i la Geltrú, 15 de setembre de 2020

2

Exercici

L'objectiu d'aquest exercici és resoldre problemes usant piles i cues implementades per nosaltres mateixos usant memòria dinàmica.

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

- Inversió separada amb piles (P13304)
- Elimina majors suma anteriors en una cua (X82400)
- Cues d'un supermercat (1) (P90861)

Els problemes d'aquesta secció no poden utilitzar les classes `stack` i `queue` de la STL. Cal incloure la definició i implementació pròpia de les classes pila i/o cua genèriques amb memòria dinàmica (la pila la trobareu en l'apèndix C i la cua en els apunts de teoria; per evitar problemes copiant des de fitxers PDF les podeu copiar de la carpeta `/home/public/esin/sessio2`). Per exemple, en el primer problema, a més a més de l'especificació i implementació de la classe `pila` (fitxers `pila.hpp` i `pila.cpp`), cal implementar el programa principal (`main.cpp`) que resol el problema.

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 el programa principal. I també cal eliminar la directiva `#include "pila.hpp"` per no tenir problemes de precompilació. Ho pots fer tot a la vegada amb la comanda:

```
cat pila.hpp pila.cpp main.cpp | sed '/include "pila.hpp"/d' > solucio.cpp
```

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

Fixa't que la classe `pila` és una pila d'enters però la classe `cua` és una classe templatitzada (plantilla) que permet fer cues de qualsevol tipus. Les classes templatitzades no es compilen, sinó que s'inclou tot el seu codi en el fitxer que les usa. Per evitar compilarla, el fitxer que conté la implementació dels mètodes té l'extensió .t (així sabem que es refereix a una plantilla o template). El fitxer `cua.hpp` ja inclou el fitxer `cua.t`. Aquesta comanda permet ajuntar els fitxers per crear una solució que jutge.org pot processar:

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

Com que possiblement aquests problemes ja els teniu resolts des de les pràctiques de PRO1 usant les classes `stack` i `queue` de la STL, envieu les noves solucions a jutge.org amb l'anotació "Fet amb la classe pila memòria dinàmica" o "Fet amb la classe cua memòria dinàmica" perquè el professor sàpiga quina versió mirar quan us la corregeixi.

C

Memòria dinàmica

C.1 Introducció

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

C.2 Reserva i alliberament de memòria dinàmica

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

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

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

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

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

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

```
// pc[i] ≡ *(pc+i) accedeix a la i-èssima component

...

delete pi;
// allibera la memòria de l'enter apuntat per pi; el punter conserva el seu
// valor, però ara apunta a una zona de memòria disponible per reservar.

delete[] pc;
// allibera la memòria de la taula de caràcters apuntada per pc
```

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

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

```

class vector {
  public:
      vector(int s = 8);          // vector de s enters
      vector(const vector &v);    // constructor per còpia
      ~vector();                  // destructor
      ...
};

int main() {
    vector* pv1 = new vector(12);
    // crea un vector de 12 enters apuntat per pv1

    vector* pv2 = new vector;
    // crea un vector de 8 enters apuntat per pv2

    vector* ptv1 = new vector[10];
    // crea una taula de 10 vectors de 8 enters apuntada per ptv1

    vector* pv3 = new vector(*pv2);
    // crea un còpia de *pv2 apuntat per pv3

    vector* ptv2 = new vector[10](*pv1);
    // crea una taula de 10 vectors on cada un dels vectors és una còpia de
    // *pv1 i està apuntada per ptv2

```

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

C.3 Problemes amb la gestió de la memòria dinàmica

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

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

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

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

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

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

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

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

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

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

O també:

```
1 class estudiant {
2   public:
3      estudiant(char* nom, int dni);
4      char* consulta_nom();
5      int consulta_dni();
6      ...
```

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

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

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

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

REGLA DELS TRES GRANS

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

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

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

C.4 Constructor per còpia

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

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

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

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

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

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

Exemple de constructor per còpia:

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

C.5 Destructor

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

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

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

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

Exemple de destructor:

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

C.6 Operador d'assignació

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

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

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

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

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

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

Exemple d'operador d'assignació:

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

C.7 El component this

this@this

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

```
a = b = 0 ;

// equival a:

a = (b = 0);

// equival a:

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

C.8 Exemple pila

C.8.1 Especificació de la classe pila

Fitxer `pila.hpp`:

```
#ifndef _PILA_HPP
#define _PILA_HPP

class pila {
  public:
      pila();                  // constructor

      // tres grans
      pila(const pila& p); // constructor per còpia
      ~pila();                 // destructor
      pila& operator=(const pila& p); // operador assignació

      void apilar(int x);
      void desapilar();
      int cim() const;
      bool es_buida() const;

  private:
      struct node { // definició de tipus privat
         node* seg; // punter al següent 'node'
         int info;
      };

      node* _cim; // la pila consisteix en un punter al node del cim

      // mètode privat de classe per alliberar memòria; allibera la cadena de
      // nodes que s'inicia en el node n.
      static void esborra_pila(node* n);

      // mètode privat de classe per realitzar còpies; còpia tota la cadena de nodes
      // a partir del node apuntat per origen i retorna un punter al node inicial de
      // la còpia; la paraula reservada const indica que no es pot modificar el valor
      // apuntat pel punter origen.
      static node* copia_pila(const node* origen);
};
#endif
```

C.8.2 Implementació de la classe pila

Fitxer `pila.cpp`:

```
#include "pila.hpp"

//-----------------
// MÈTODES PRIVATS DE CLASSE
//-----------------

void pila::esborra_pila(node* n) {
    if (n != NULL) {
        esborra_pila(n->seg); // p->seg equival a (*p).seg
        delete n; // allibera la memòria de l'objecte apuntat per n.
    }
}

// és necessari posar pila::node com tipus del resultat per què node
// està definit de la classe pila
pila::node* pila::copia_pila(const node* origen) {
    node* desti = NULL
    if (origen != NULL) {
        desti = new node;
        desti->info = origen->info;

        // copia la resta de la cadena
        desti->seg = copia_pila(origen->seg);
    }
    return desti;
}

//-----------------
// MÈTODES PÚBLICS
//-----------------
pila::pila() : _cim(NULL) { }

// genera una còpia de la pila apuntada per 'p._cim'
pila::pila(const pila& p) {
    _cim = copia_pila(p._cim);
}

// allibera la memòria de la pila apuntada per '_cim'
pila::~pila() {
    esborra_pila(_cim);
}

```

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

C.8.3 Programa d'exemple que usa la classe pila

Fitxer `exemple_pila.cpp`:

```
1 #include <iostream>
2 #include "pila.hpp"
3
4 using namespace std;
5
```

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