Normes bàsiques de programació en C++

Salvador Roura

1 de setembre de 2010

1 Preàmbul

Aquestes normes es van redactar inicialment per a l'assignatura "Programació 1" de la FIB, on es van aplicar durant quatre anys (cursos 2006–2007 al 2009–2010), però estan pensades per a qualsevol curs d'aprenentatge de la programació en C++.

L'estil de programació proposat en aquest document busca un difícil compromís: D'una banda, cal evitar que un curs introductori es converteixi en un curset de les moltes instruccions i construccions que els llenguatges de programació moderns ofereixen. És a dir, cal ensenyar a l'alumne de manera que aquest no acabi sent totalment dependent del seu primer llenguatge de programació, i pugui adaptar-se ràpidament a altres llenguatges. D'altra banda, és bo que els alumnes aprenguin des del primer moment a fer fàcil allò que és fàcil, cosa que a vegades requereix l'ús de construccions més o menys específiques del llenguatge de programació escollit.

Aquestes normes van ser redactades després de consultar força material on-line, en particular el de les adreces referenciades al final d'aquest document, i intentant en tot moment seguir les recomenacions majoritàries abans que el gust personal de l'autor. A més, es va mirar d'incloure regles molt clares i simples, i de minimitzar-ne les excepcions tant com fos possible. El resultat de tot plegat és un estil de programació "neutre", en el sentit que seguir-lo estrictament produeix codi (quasi) indistingible del d'alguns programadors excel·lents amb molts anys d'experiència.

Hi ha molts més motius que justifiquen l'existència d'unes regles estrictes en una assignatura d'ensenyament bàsic de la programació. Per exemple, un estil fixat evita que els estudiants malgastin energies decidint com escriuen el seu codi, i que agafin mals hàbits que després costa eliminar. Addicionalment, un estil consistent ajuda a reduir els errors. A més, amb un estil comú, els estudiants comprenen millor les solucions dels exercicis, ja sigui dels professors o d'altres estudiants, perquè estan presentades de forma més semblant a les seves. Similarment, els professors poden entendre millor què fan (o què pretenen fer) els codis dels estudiants.

Finalment, l'experiència de P1 va constatar un altre efecte beneficiós: molts estudiants van usar aquest estil com un estàndard durant la resta de la carrera, cosa que els va facilitar les pràctiques en grup, per exemple.

Com a nota anecdòtica, aquestes van ser les primeres regles estrictes que es van imposar a la FIB en una assignatura d'ensenyament de la programació.

2 Algunes consideracions generals

(Totes les afirmacions que segueixen s'han d'interpretar només en el context d'una assignatura d'aprenentatge de la programació, i no de forma universal.)

- Els programes escrits en C++ tenen l'extensió `.cc`.
- Cada línia de codi té una única instrucció bàsica.

  Exemple:

  ```
  i = 0;
  j = 1;
  ```

  és millor que

  ```
  i = 0; j = 1;
  ```

  Però

  ```
  i = j = 0;
  ```

  és millor que

  ```
  i = 0;
  j = 0;
  ```

- Una variable es declara quan es necessita, mai abans, i sovint s'inicialitza en la pròpia declaració. En aquest cas, cal declarar-la individualment.

  Exemple:

  ```
  double distancia(double x1, double y1, double x2, double y2) {
      double x = x1 - x2;
      double y = y1 - y2;
      return sqrt(x*x + y*y);
  }
  ```

  és millor que

  ```
  double distancia(double x1, double y1, double x2, double y2) {
      double x, y;
      x = x1 - x2;
      y = y1 - y2;
      return sqrt(x*x + y*y);
  }
  ```

  i que

  ```
  double distancia(double x1, double y1, double x2, double y2) {
      double x = x1 - x2, y = y1 - y2;
      return sqrt(x*x + y*y);
  }
  ```

- Algunes situacions típiques en què no s'inicialitza una variable en la pròpia declaració són:
  - Per llegir.
    Exemple:

    ```
    int x, y;
    cin >> x >> y;
    ```

  - Quan el valor depèn d'una condició.
    Exemple:

    ```
    double segons;
    if (graus) segons = 3600*angle;
    else segons = 60*angle;
    ```

  - Quan el valor depèn d'un bucle.
    Exemple:

    ```
    // Volem sumar tots els nombres i saber la posicio de qualsevol
    // que sigui negatiu. Sabem que n'hi ha un, com a minim.
    int posicio_negatiu;
    double suma = 0;
    for (int i = 0; i < v.size(); ++i) {
        suma += v[i];
        if (v[i] < 0) posicio_negatiu = i;
    }
    ```

  - Quan s'usarà immediatament com a paràmetre de sortida.
    Exemple:

    ```
    vector<double> v;
    llegeix_vector(v);
    double suma, producte;
    // Calcula la suma i el producte de tots els elements de v.
    suma_i_multiplica(v, suma, producte);
    ```

  - En algun cas extrem com ara:

    ```
    int i1 = 0;
    int i2 = 0;
    int i3 = 0;
    int i4 = 0;
    int i5 = 0;
    int i6 = 0;
    int i7 = 0;
    int i8 = 0;
    ```

    Aquí, potser és millor estalviar línies fent

    ```
    int i1, i2, i3, i4, i5, i6, i7, i8;
    i1 = i2 = i3 = i4 = i5 = i6 = i7 = i8 = 0;
    ```

- Les variables s'han de declarar dins de l'àmbit de visibilitat més intern possible.

  Exemple:

```
int x, y;
cin >> x >> y;
if (x > y) {
    int aux = x;
    x = y;
    y = aux;
}
cout << "elements ordenats: " << x << " " << y << endl;
```

  és millor (i també diferent) que

```
int x, y, aux;
cin >> x >> y;
if (x > y) {
    aux = x;
    x = y;
    y = aux;
}
cout << "elements ordenats: " << x << " " << y << endl;
```

  En el segon cas, la variable auxiliar `aux` segueix existint després de l'escriptura. Això podria provocar col·lisions de noms, si més endavant es vol usar una altra variable amb nom `aux`.

- La regla anterior té un cas particularment important: la variable de control d'un `for`, quan el seu contingut no és necessari un cop acabat el bucle. En aquest cas, molt habitual, la declaració de la variable s'ha de fer dins del `for`.

  Exemple:

```
for (int i = 0; i < 10; ++i) cout << i*i << endl;
```

  és millor (i també diferent) que

```
int i;
for (i = 0; i < 10; ++i) cout << i*i << endl;
```

  Només en el segon cas la variable `i` segueix existint després del `for`. Si, unes quantes línies de codi més avall, volguéssim fer un altre bucle usant també la variable `i`, hauríem de recordar que ja està declarada, per no redeclarar-la i patir un error de compilació. En canvi, aquest codi compila sense problemes:

```
for (int i = 0; i < 10; ++i) cout << i*i << endl;
for (int i = 0; i < 10; ++i) cout << i*i*i << endl;
```

- Convé usar una sola variable de control per a cada `for`.

  Exemple:

```
vector<double> invers(const vector<double>& v) {
    int n = v.size();
    vector<double> u(n);
    for (int i = 0; i < n; ++i) u[i] = v[n - 1 - i];
    return u;
}
```

  és millor que

```
    ...
    for (int i = 0, j = n - 1; i < n; ++i, --j) u[i] = v[j];
    ...
```

  Si cal, sovint es pot recuperar el valor de les altres variables de control a cada iteració (no en aquest exemple, on s'obté un codi innecessàriament verbós):

```
    ...
    for (int i = 0; i < n; ++i) {
        int j = n - 1 - i;
        u[i] = v[j];
    }
    ...
```

- Si alguna de les tres parts de la "capçalera" d'un `for` hagués de ser buida, o si calgués modificar la variable de control d'un `for` en el cos del bucle, llavors és millor usar un `while`.

  Exemple:

```
// Retorna la primera posicio de v on hi ha x, o -1 si x no hi es.
int posicio(double x, const vector<double>& v) {
    bool trobat = false;
    int i = 0;
    while (not trobat and i < v.size()) {
        if (v[i] == x) trobat = true;
        else ++i;
    }
    if (trobat) return i;
    else return -1;
}
```

  és millor que

```
bool posicio(double x, const vector<double>& v) {
    bool trobat = false;
    int i = 0;
    for (; not trobat and i < v.size();) {
        if (v[i] == x) trobat = true;
        else ++i;
    }
    if (trobat) return i;
    else return -1;
}
```

- A vegades, posar un `return` dins d'un bucle és la manera més senzilla i fiable de programar, i produeix codi més eficient i llegible.

  Per exemple, aquest codi és millor que els dos codis anteriors:

```
int posicio(double x, const vector<double>& v) {
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

- Cal evitar que variables d'àmbits interns amaguin altres variables d'àmbits més externs. Encara que el codi fos correcte, seria molt confús.

  Exemple:

```
// Dibuixa un quadrat nxn.
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) cout << "*";
    cout << endl;
}
```

  és millor que

```
for (int i = 0; i < n; ++i) {
    for (int i = 0; i < n; ++i) cout << "*";
    cout << endl;
}
```

- No s'usen "nombres màgics". En lloc seu, cal fer servir constants.

  Exemple:

```
const int MAX_PERSONES = 700;
...
if (n > MAX_PERSONES/2) cout << "Esta bastant ple." << endl;
```

  és millor que

```
if (n > 350) cout << "Esta bastant ple." << endl;
```

- Cal evitar tant com es pugui fer comparacions d'igualtat o de desigualtat entre nombres reals, especialment si aquests són el resultat d'una sèrie d'operacions, ja que petits errors d'arrodoniment poden fer fallar el programa.

- No es compara mai un booleà amb `true` o amb `false`.

  Exemple:

  ```
  if (trobat) ...
  if (not repetit) ...
  ```

  és millor que

  ```
  if (trobat == true) ...
  if (repetit == false) ...
  ```

- Les expressions booleanes en C++ s'avaluen d'esquerra a dreta i parant quan es coneix el resultat. Això es pot aprofitar per simplificar el codi.

  Exemple:

  ```
  int i = 0;
  while (i < v.size() and v[i] != x) ++i;
  if (i < v.size()) cout << x << " es troba a " << i << endl;
  ```

  és millor que

  ```
  bool trobat = false;
  int i = 0;
  while (not trobat and i < v.size()) {
      if (v[i] == x) trobat = true;
      else ++i;
  }
  if (trobat) cout << x << " es troba a " << i << endl;
  ```

△ *Està prohibit usar referències fora del pas de paràmetres.*

△ *Els punters estan prohibits.*

△ *Les paraules clau* `new`, `delete`, `goto`, `continue`, `break`, `switch`, `try`, `throw`, `catch`, `template` *i* `operator` *estan prohibides.*

△ *Està prohibit fer crides a sistema.*

3 Construccions no recomenades en un curs introductori[1]

△ Els preincrements amb efectes laterals.

Per exemple, aquest codi és pitjor que l'últim codi recomenat:

```
int i = -1;
while (++i < v.size() and v[i] != x);
if (i < v.size()) cout << x << " es troba a " << i << endl;
```

△ Els postincrements: `i++; j--;`

△ La construcció `do while`.

△ Els operadors sobre bits: `&`, `|`, `<<`, `>>`, `^`.

△ La construcció *condició_booleana* `?` *expressió1* `:` *expressió2*.

△ Els operadors booleans obsolets `&&`, `||`, `!`.

En lloc seu, cal usar els moderns `and`, `or`, `not`.

△ La conversió de tipus de C.

En els pocs casos en què s'hagi de convertir d'un tipus a un altre, convé fer-ho explícitament, i usant la notació moderna de C++.

Exemple (friqui):

```
int i = int(M_PI);
cout << char('a' + i);
```

és millor que

```
int i = (int)M_PI;
cout << (char)('a' + i);
```

△ La comparació implícita amb el valor nul del tipus (excepte amb els booleans).

Exemple:

```
int x, y;
cin >> x >> y;
if (x == 0) cout << "el primer es zero" << endl;
if (y != 0) cout << "el segon no es zero" << endl;
```

és millor que

```
...
if (!x) cout << "el primer es zero" << endl;
if (y) cout << "el segon no es zero" << endl;
```

△ Els prototipus, és a dir, declarar els procediments abans de la seva definició. No calen perquè no hi ha recursivitat encreuada.

[1] Algunes d'aquestes construccions no presenten cap inconvenient per als programadors experts.

4 Exemple de programa

```
#include <iostream>
#include <vector>
using namespace std;


// Nombre maxim de camins que es poden llegir.
const int MAX_CAMINS = 100;


// Un punt en el pla te una component x i una component y.
struct Punt {
    double x, y;
};


// Un cami es una sequencia de punts.
typedef vector<Punt> Cami;


// Llegeix i retorna un punt.
Punt punt_llegit() {
    Punt p;
    cin >> p.x >> p.y;
    return p;
}


// Llegeix un cami, del qual es dona primer n, i despres els n punts.
// n = 0 marca el final de l'entrada. En aquest cas, final es posa a cert.
void llegeix_cami(Cami& c, bool& final) {
    int n;
    cin >> n;
    if (n == 0) final = true;
    else {
        c = Cami(n);
        for (int i = 0; i < n; ++i) c[i] = punt_llegit();
    }
}

...


// Llegeix un maxim de MAX_CAMINS.
// A continuacio, ...
int main() {
    vector<Cami> v(MAX_CAMINS);
    bool final = false;
    int n = 0;
    llegeix_cami(v[n], final);
    while (not final) {
        ++n;
        llegeix_cami(v[n], final);
    }
    ...
}
```

5 Estructura d'un programa

Com es pot apreciar a l'exemple anterior, els programes tenen cinc apartats:

1. Inclusions i espai de noms
2. Definició de constants—pot ser buit
3. Definició de nous tipus (vectors i structs)—pot ser buit
4. Procediments—pot ser buit
5. Programa principal (`main`)

5.1 Inclusions i espai de noms

Només es posen les inclusions necessàries, amb la directiva `#include`.

△ *Excepte* `#include`*, totes les altres directives estan prohibides.*

Les úniques inclusions admeses són:

- `<iostream>`: Només es pot usar la construcció

  `cin >>` *variable1* `>>` *variable2 ...* `;`

  per llegir, i la construcció

  `cout <<` *expressió1* `<<` *expressió2 ...* `;`

  per escriure.

  Si cal escriure reals, primer s'ha de decidir el nombre d de decimals. Després, cal incloure les dues línies següents al principi del `main`:

  ```
  cout.setf(ios::fixed);
  cout.precision(d);
  ```

  Si es vol un salt de línia, cal escriure `endl` (i no `'\n'`, ni res semblant).

  Hi ha tres esquemes típics de bucle de lectura:

  1. coneixent el nombre d'elements a priori,
  2. fins a trobar un centinella,
  3. fins que l'entrada s'acabi.

  Els dos primers es codifiquen trivialment. La bona manera d'escriure el tercer esquema és usant la conversió implícita de `cin` a booleà.

Exemple:

```
int suma = 0;
int x;
while (cin >> x) suma += x;
cout << "la suma de tots val " << suma << endl;
```

Cal entendre "`while (cin >> x)`" com "mentre hagi pogut llegir `x`".

Usar `cin.eof()` és una font potencial d'errors.

△ *Està prohibit usar altres operacions d'entrada/sortida com ara* `getline()`.

△ *Està prohibida la lectura i escriptura a l'estil de C, amb* `scanf()`, `printf()` *o similars.*

- `<string>`: Es poden declarar (sense valor inicial, o bé assignant-los un literal), assignar, comparar lexicogràficament, llegir i escriure.

  Exemples:

```
const string SALUTACIO = "Hola!!!";

...

cout << SALUTACIO << endl;
string s, t;
cin >> s >> t;
if (s > t) {
    string aux = s;
    s = t;
    t = aux;
}
cout << s << " es mes petit o igual que " << t << endl;
```

  A més, a partir de mig curs, els strings es poden declarar donant la mida i el valor inicial a cada posició, es pot consultar la seva mida, i es pot accedir a cadascuna de les seves posicions.

  Exemples:

```
string s;
cin >> s;
cout << s << " te " << s.size() << " caracter(s)" << endl;
cout << "el primer caracter es " << s[0] << endl; // suposant que s no es buit

...

int n;
char c;
cin >> n >> c;
string t(n, c);
cout << t << " te " << n << " caracter(s)" << endl;
cout << "tots iguals a " << c << endl;
```

Cal tenir cura amb les operacions aritmètiques amb diverses crides a `size()` (tant de strings com de vectors), perquè `size()` retorna enters sense signe, els quals combinats (en particular, restats) poden donar resultats incorrectes.

Per exemple:

```
string x, y;
cin >> x >> y;
if (x.size() - y.size() > 0) cout << "x es mes llarg que y" << endl;
```

Aquest codi *no* és correcte. (Proveu què passa quan `x` és més curt que `y`.) Afortunadament, sovint es pot reescriure el codi i fer-lo correcte:

```
string x, y;
cin >> x >> y;
if (x.size() > y.size()) cout << "x es mes llarg que y" << endl;
```

△ *No es pot usar cap altra operació sobre strings.*

- `<vector>`: Es poden declarar (amb o sense mida inicial i, en el primer cas, amb o sense valor inicial a cada posició) i assignar. A més, es pot consultar la seva mida i accedir-ne a cadascuna de les posicions.

  Exemples:

```
void llegeix_vector(vector<double>& v) {
    int n;
    cin >> n;
    v = vector<double>(n);
    for (int i = 0; i < n; ++i) cin >> v[i];
}


void dobla(vector<int>& v) {
    for (int i = 0; i < v.size(); ++i) v[i] *= 2;
}

....

vector<double> v;
llegeix_vector(v);
vector<int> u(100, -4);
dobla(u); // ara u es igual a un vector<int>(100, -8)
```

  △ *Està prohibit usar vectors en els exercicis on no són realment necessaris.*

  △ *No es pot usar cap altra operació sobre vectors.*

- `<cmath>`: Només es permet usar la constant predefinida `M_PI` (π amb la millor precisió permesa per la màquina), les funcions trigonomètriques `sin()`, `cos()` i `tan()`, i l'arrel quadrada `sqrt()`.

  △ *No es pot usar cap altra operació matemàtica.*

- `<algorithm>`: En els exercicis de final de curs que no diguin explícitament el contrari, si cal ordenar un vector es pot usar el procediment estàndard `sort()`.

  Exemple:

  ```
  vector<int> v(MAX);
  for (int i = 0; i < MAX; ++i) cin >> v[i];
  sort(v.begin(), v.end());
  cout << "ordenats:";
  for (int i = 0; i < MAX; ++i) cout << " " << v[i];
  cout << endl;
  ```

  Si es vol ordenar un vector els elements del qual no són bàsics, o amb algun criteri diferent de l'habitual (de petit a gran), es pot fer fàcilment usant una funció de comparació que indiqui quan un element és més petit que un altre.

  Exemple:

  Suposem que el socis d'un club es defineixen amb diversos camps, dos dels quals són el nom i l'edat:

  ```
  struct Soci {
      string nom;
      int edat;
      ...
  };
  ```

  Si volem ordenar els socis primer per edat de més vells a més joves, i en cas d'empat creixentment per nom, llavors implementem aquesta funció:

  ```
  bool comp(const Soci& a, const Soci& b) {
      if (a.edat != b.edat) return a.edat > b.edat;
      return a.nom < b.nom;
  }
  ```

  Ara, per ordenar un vector de socis `v` n'hi hauria prou de fer:

  ```
  sort(v.begin(), v.end(), comp);
  ```

  Fixeu-vos que aquesta funció de comparació *no* és correcta, perquè en cas que `a` i `b` siguin iguals, indica que `a` hauria d'anar abans que `b`, cosa que és falsa:

  ```
  bool comp(const Soci& a, const Soci& b) {
      if (a.edat != b.edat) return a.edat > b.edat;
      return a.nom <= b.nom;
  }
  ```

  △ *No es pot usar cap altra operació de la llibreria* `<algorithm>`.

A continuació de les inclusions, cal afegir sempre la línia

```
using namespace std;
```

△ *Està prohibit usar qualsevol altre espai de noms, o qualsevol inclusió que no sigui* `<iostream>`, `<string>`, `<vector>`, `<cmath>` *o* `<algorithm>`.

5.2 Definició de constants

Cada constant s'ha de definir en una línia apart, usant la sintaxi

`const` *nom_de_tipus nom_de_constant* `=` *valor*`;`

La paraula clau `const` fa que es defineixi una constant i no una variable global. Una variable és global si està declarada fora de tots els procediments, el `main` inclòs.

△ *Les variables globals estan prohibides.*

5.3 Definició de nous tipus

Els únics tipus bàsics que es poden usar són: `int`, `double`, `bool` i `char`. El tipus `string` no és bàsic; cal incloure la llibreria `string` per usar-lo.

△ *Està prohibit usar qualsevol altre tipus bàsic o modificador. Per exemple, no es pot usar* `short (int)`*,* `long (int)`*,* `long long (int)`*,* `unsigned int` *o* `float`*.*

△ *Els tipus enumerats (*`enum`*) estan prohibits.*

△ *Els arrays a l'estil de C (declarats amb* `[]`*) estan prohibits.*

△ *Està prohibit usar* `class` *per definir nous tipus.*

5.4 Procediments

Els paràmetres d'entrada es passen per valor (és a dir, per còpia) en el cas dels tipus bàsics i strings, i per referència constant altrament (vectors i structs).[2] Els paràmetres de sortida o d'entrada/sortida es passen per referència.

Hi ha dos tipus de procediments: accions i funcions.

- Accions: Les accions no retornen res, cosa que es marca amb `void`.

  Exemple:

```
void suma_i_multiplica(const vector<double>& v, double& suma,
                       double& producte) {
    suma = 0;
    producte = 1;
    for (int i = 0; i < v.size(); ++i) {
        suma += v[i];
        producte *= v[i];
    }
}
```

  Sovint és una bona pràctica posar els paràmetres en aquest ordre: entrada, entrada/sortida, sortida.

[2] Aquí, estem suposant que els strings usats en els exercicis del curs són "prou petits". Altrament, caldria passar-los seguint la mateixa convenció dels vectors.

- Funcions: Les funcions retornen qualsevol tipus (preferentment bàsic o string, per qüestions d'eficiència), i tenen tots els paràmetres d'entrada.

  Exemple:

```
double producte(const vector<double>& v) {
    double prod = 1;
    for (int i = 0; i < v.size(); ++i) prod *= v[i];
    return prod;
}
```

Com es pot veure a l'exemple anterior, per evitar que un nom n'amagui un altre, és bo donar a les variables noms diferents dels noms dels procediments. Això és imprescindible en els procediments recursius.

D'alguns exemples es pot veure com els operadors `+=`, `-=`, `*=`, `/=`, i `%=` estan permesos. De fet, se n'encoratja l'ús quan no sigui forçat.

5.5 Programa principal

El programa principal té sempre la capçalera

```
int main();
```

Encara que el `main` ha de retornar un enter que indica l'estat final del programa (0, si el programa ha anat bé), el compilador entén que hi ha implícitament la instrucció `return 0;` al final del programa. Per tant, no cal posar-la mai. Addicionalment, cal fixar-se que el nostre `main` no té cap paràmetre.

6 Noms de constants, tipus, procediments i variables

Si un nom està format per diverses paraules, aquestes se separen amb un '_'. La resta són lletres majúscules i minúscules (i, a vegades, algun dígit en els noms de variables), amb la convenció següent:

- Els noms de constants s'escriuen exclusivament amb lletres majúscules:

```
const double PROPORCIO_DIVINA = (1 + sqrt(5.0))/2;
const int MAX_PERSONES = 1000;
```

- Cada paraula del nom d'un tipus comença amb una lletra majúscula, seguida de lletres minúscules:

```
typedef vector<int> Fila;
typedef vector<Fila> Matriu;


struct Punt_Tridimensional {
    double x, y, z;
};
```

- Els noms de procediments i de variables s'escriuen exclusivament amb lletres minúscules.

Cal escollir sempre noms significatius per a les constants, tipus i procediments. En particular, no ser capaç de trobar un bon nom per a una constant o un procediment és senyal que la constant o el procediment no estan prou ben meditats.

Respecte a les variables, a vegades és preferible usar noms curts per augmentar la llegibilitat, com es pot apreciar en molts exemples anteriors (això és particularment cert per als índexos dels vectors).

A més, cal evitar posar noms negats a les variables booleanes. Exemple:

```
bool trobat = false;
```

o bé

```
bool cal_seguir = true;
```

és millor que

```
bool no_trobat = true;
```

És convenient usar aquestes regles per als noms dels procediments:

- Els noms de les accions són (o inclouen) verbs en forma imperativa.

  Exemples:

  - `void ordena(vector<double>& v);`
  - `void elimina_repetits(vector<string>& v);`
  - `void fusiona(vector<int>& v1, vector<int>& v2);`

- Les funcions que retornen un booleà comencen habitualment amb `"es_"` o `"esta_"`.

  Exemples:

  - `bool es_primer(int n);`
  - `bool es_digit(char c);`
  - `bool esta_ordenat(const vector<string>& v);`

  Però també hi ha altres possibilitats:

  - `bool pertany(int a, const vector<int>& v);`

- El nom de les funcions que retornen tipus no booleans solen ser substantius que indiquen què es retorna.

  Exemples:

  - `int factorial(int n);`
  - `int nombre_de_repetits(const vector<double>& v);`
  - `string maxim(const vector<string>& v);`

  Però també:

  - `char a_minuscula(char c);`

7 Línies en blanc, indentació, espais, claus i parèntesis

7.1 Línies en blanc

Convé separar amb (dues, per exemple) línies en blanc els diferents apartats d'un programa (inclusions, constants, tipus, procediments i `main`). Dins de cada apartat, convé separar cadascun dels seus subapartats amb els mateix nombre de línies en blanc, amb dues excepcions: totes les declaracions de constants i totes les definicions de tipus amb `typedef` s'escriuen sovint sense cap separació (al principi de la secció 6 hi ha un exemple de cada).

A vegades, quan un tros de codi és llarg, és bo posar una línia en blanc que en separi dos blocs conceptuals. Si el codi és excessivament llarg, però, el que cal és fer-lo més llegible usant procediments auxiliars.

7.2 Indentació

- S'indenta amb quatre espais per nivell.
- Cal evitar els tabuladors, configurant l'editor per desactivar-los.
- Les línies de més de 78 caràcters s'han de trencar, ja que sovint no es poden imprimir bé.
- Si un tros de codi té massa indentacions, cal fer-lo més llegible usant algun procediment auxiliar.

7.3 Espais

Com a norma general, cal deixar exactament un espai de separació entre les paraules clau, instruccions, literals, etcètera. Les excepcions més importants són:

- Els punts i coma s'escriuen enganxats a allò que tinguin a la seva esquerra.
- La part interna d'un parèntesi no se separa d'allò que contingui.
- El parèntesi esquerre d'un procediment s'escriu enganxat al seu nom.
- Amb les claus d'accés a una posició d'un vector o string se segueix exactament el mateix espaiat que amb els parèntesis d'un procediment.
- Els operadors binaris `*` (producte), `/` (divisió) i `%` (residu), i l'operador unari `-` (canvi de signe) s'escriuen sense espais per emfasitzar la seva alta precedència.

  Exemple:

  ```
  int a = x + y*z;
  ```

  és millor que

  ```
  int a = x + y * z;
  ```

  i que

  ```
  int a = x+y*z;
  ```

També:

```
int a = -b;
```

és millor que

```
int a = - b;
```

- Els operadors `++` (preincrement) i `--` (predecrement) s'escriuen a l'esquerra de la variable que modifiquen, sense cap espai.
- El caràcter `&` (per indicar que un paràmetre es passa per referència) s'escriu immediatament a la dreta del tipus al qual està associat.

Aquest exemple inclou molts dels punts anteriors:

```
void ves_a_saber_qui_voldria_fer_aixo(const vector<int>& v) {
    for (int i = v.size() - 5; i >= -4; --i) {
        cout << -i << " " << factorial(i*i + 3)/v[i + 4] - 7 << endl;
    }
}
```

7.4 Claus

Hi ha moltes maneres i variants de posar les claus. La que es recomana aquí és:

```
if (condicio) {
    ...
}

if (condicio) {
    ...
}
else {
    ...
}

while (condicio) {
    ...
}

for (inicialitzacio; condicio; "increment") {
    ...
}
```

Com es pot veure, no es posa mai res a la dreta d'una clau, ja sigui d'obrir o de tancar.

A vegades es té una col·lecció de condicions exclusives que cal avaluar una rera l'altra. Això dóna lloc a una estructura `if else if else if ...`, la qual cal escriure així:

```
if (condicio1) {
    ...
}
else if (condicio2) {
    ...
}
else if (condicio3) {
    ...
}
...
else {
    ...
}
```

Exemple:

```
string tipus(char c) {
    if (c >= 'a' and c <= 'z') {
        return "minuscula";
    }
    else if (c >= 'A' and c <= 'Z') {
        return "majuscula";
    }
    else if (c >= '0' and c <= '9') {
        return "digit";
    }
    else {
        return "desconegut";
    }
}
```

Quan dintre del cos d'un `if`, d'un `while` o d'un `for` només es realitza una acció, i aquesta cap a la mateixa línia, es pot evitar posar les claus per fer un codi més curt i compacte:

```
for (int i = 0; i < v.size(); ++i) prod *= v[i];
```

Fins i tot, els `return` permeten estalviar els `else`:

```
string tipus(char c) {
    if (c >= 'a' and c <= 'z') return "minuscula";
    if (c >= 'A' and c <= 'Z') return "majuscula";
    if (c >= '0' and c <= '9') return "digit";
    return "desconegut";
}
```

7.5 Parèntesis

- Com ja s'ha dit, els parèntesis se separen de les paraules clau, la part interna d'un parèntesi no se separa d'allò que contingui, i el parèntesi esquerre d'un procediment s'escriu immediatement a continuació del nom del procediment.

- No es posen parèntesis al voltant d'allò que una funció retorna.

  Exemple:

  ```
  return trobat;
  ```

  és millor que

  ```
  return (trobat);
  ```

- Com a norma general, només es posen els parèntesis estrictament necessaris per canviar la precedència dels operadors.

  Exemple:

  ```
  int x = a + b*c;
  ```

  és millor que

  ```
  int x = a + (b*c);
  ```

- En els casos en què la precedència no és òbvia a la vista (per exemple, quan caldria recordar si les operacions de la mateixa prioritat s'avaluen d'esquerra a dreta o a l'inrevés), és convenient afegir parèntesis per augmentar la llegibilitat.

  Exemple:

  ```
  int a = (x/y)*z;
  ```

  o bé (segons calgui)

  ```
  int a = x/(y*z);
  ```

  és millor que

  ```
  int a = x/y*z;
  ```

- Excepcionalment, si una expressió és tan complicada que costa llegir-la sense parèntesis redundants, aquests es poden afegir. En aquests casos, però, sovint és millor usar variables intermitges per simplificar les expressions.

8 Comentaris

Cal documentar el codi en la justa mesura. Posar comentaris redundants pot ser tan perniciós com ometre els comentaris importants. En concret, cal explicar en unes poques línies *què* fa cada procediment. En els casos en què no sigui trivial *com* ho fa, cal afegir un comentari breu explicant-ho.

Exemples[3]:

```
//  Retorna n!.
//  Pre: n >= 0.
int factorial(int n) {
    if (n == 0) return 1;
    else return n*factorial(n - 1);
}


//  Intercanvia a i b.
void intercanvia(int& a, int& b) {
    int aux = a;
    a = b;
    b = aux;
}


// Diu si n es capicua.
// Pre: n >= 0, p es la potencia de 10 tal que p <= n < 10*p,
//      o be es qualsevol valor si n = 0.
// Explicació:
// Un nombre amb un sol digit es capicua.
// Altrament, comprovem que el primer i l'ultim digit siguin iguals,
// i comprovem recursivament que la resta del nombre sigui capicua.
// Cal mantenir l'invariant de p, dividint-lo per 100.
bool es_capicua(int n, int p) {
    if (n <= 9) return true;
    int primer_digit = n/p;
    int ultim_digit = n%10;
    return primer_digit == ultim_digit
           and es_capicua((n - primer_digit*p)/10, p/100);
}
```

Com es pot veure, per documentar s'usa `//`. Ens reservem `/* */` per a la fase de depuració de programes. Així es poden compilar només certs trossos de codi, esborrant lògicament (però no físicament) els trossos que no es volen compilar. El programa final no ha d'incloure codi "esborrat" amb `/* */`.

Un programa tampoc no pot tenir codi al qual no es pugui arribar mai. Un exemple típic d'això és el següent: Volem fer una funció que retorni el signe d'un nombre, i escrivim:

```
int signe(int a) {
    if (a > 0) return 1;
    else if (a < 0) return -1;
    else if (a == 0) return 0;
}
```

[3]Possiblement, amb massa comentaris.

El compilador no entén que les tres condicions són excloents, i ens dóna l'avís, que a vegades es tracta com un error, que la funció pot ser que no retorni res. La mala solució per corregir-ho és afegir una línia arbitrària al final (amb o sense `else`):

```
int signe(int a) {
    if (a > 0) return 1;
    else if (a < 0) return -1;
    else if (a == 0) return 0;
    else return 33;
}
```

La bona solució és treure la comprovació redundant (els dos `else` també es podrien treure, és qüestió de gustos):

```
int signe(int a) {
    if (a > 0) return 1;
    else if (a < 0) return -1;
    else return 0;
}
```

Referències

- http://www.chris-lott.org/resources/cstyle/CppCodingStandard.html
- http://www.research.att.com/~bs/JSF-AV-rules.pdf
- http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/C++STYLE
- http://www.horstmann.com/bigcpp/styleguide.html
- http://www.spelman.edu/~anderson/teaching/resources/style/
- http://geosoft.no/development/cppstyle.html
- http://geosoft.no/development/cpppractice.html