`assert`: una eina per a la depuració de programes

11 de març de 2016

Un cop disposem d'un fitxer executable, sovint es pot donar el cas que no funcioni correctament. Normalment, això es deu a que una o més variables del programa prenen valors incorrectes o no compleixen propietats que haurien de complir en un moment determinat.

Existeix una sèrie d'eines que ens faciliten la detecció i correcció (també anomenada, amb més propietat, *depuració*) d'aquests errors. Algunes es fan servir de manera senzilla i automàtica, com ara l'opció de compilació `-D_GLIBCXX_DEBUG`, però aquesta només serveix per detectar l'incompliment de precondicions en crides a operacions de classes estàndar (`vector`, `list`, etc.) També està la classe `Exception`, que permet gestionar els estats anòmals del programa, retornar un missatge descriptiu amb tota la informació que es vulgui donar i, fins i tot, definir cursos d'acció alternatius. Una vegada definida una excepció, però, no es pot desactivar fàcilment.

A continuació, presentem una altra d'aquestes eines: la macro `assert`. Amb una mica d'esforç addicional per la nostra part, ens proporcionarà una capa més de control sobre els estats dels nostres programes (no només precondicions), amb els consegüents avantatges per a la depuració. A més, podrem activar-la o desactivar-la quan vulguem.

Per fer-la servir hem de fer la inclusió

```
#include <cassert>
```

La sintaxi és molt simple, té un únic argument que és una expressió. Si en avaluar-la el resultat és fals, s'escriu un missatge i el programa avorta. El missatge pot tenir un format diferent depenent de la implementació d'`assert`, però ha d'incloure l'expressió que s'hauria de complir (i que no es compleix), el nom del fitxer font i el nombre de línia on hi havia l'`assert` fallit.

Veiem un parell d'exemples d'ús de l'`assert`. Recordem l'exercici de la suma de matrius de la sessió 1.

```
Matriz sumar_Matriz_int(const Matriz& m1, const Matriz& m2)
{
  int fil = m1.size();
  int col = m1[0].size();
```

```
  Matriz suma(fil, vector<int>(col));
  for (int i = 0; i < fil; ++i) {
    for (int j = 0; j < col; ++j) suma[i][j] = m1[i][j] + m2[i][j];
  }
  return suma;
}
```

Ens pot interessar comprovar que les dues matrius tinguin la mateixa dimensió per poder fer la suma. Això ho podem aconseguir afegint el seguent `assert` com a segona línia del codi.

```
assert(fil == m2.size() and col == m2[0].size());
```

Si l'`assert` falla, sabrem que les dimensions no coincideixen. Si volem distingir quina dimensió està malament, haurem de fer dos `assert`

```
assert(fil == m2.size());
assert(col == m2[0].size());
```

El codi quedaria així:

```
Matriz sumar_Matriz_int(const Matriz& m1, const Matriz& m2)
{
  int fil = m1.size();
  int col = m1[0].size();
  assert(fil == m2.size());
  assert(col == m2[0].size());
  Matriz suma(fil, vector<int>(col));
  for (int i = 0; i < fil; ++i){
    for (int j = 0; j < col; ++j) suma[i][j] = m1[i][j] + m2[i][j];
  }
  return suma;
}
```

Penseu ara quina mena d'`assert` podríeu fer servir per l'exercici de la multiplicació de matrius.

No sempre és possible trobar una expressió senzilla per fer la comprovació que volem. Per exemple, imagineu que tenim un vector on hem anant inserint i esborrant elements i ens interessa comprovar que en un determinat punt estigui ordenat ascendentment. Llavors podríem fer:

```
...
for (i = 1; i < v.size(); ++i) assert(v[i] > v[i-1]);
```

Una altra possibilitat podria ser fer servir una cerca normal i avaluar el booleà `trobat`.

```
...
bool trobat = false;
int i = 1;
while (i < v.size() and not trobat) {
  if (v[i] < v[i-1]) trobat = true;
  else ++i;
}
assert(not trobat);
```

Si volem afegir un missatge nostre es pot fer així:

```
assert(Expressio and "El meu missatge")
```

tot i que no és gaire necessari donat que si el programa aborta, sabrem el nombre de línia on s'ha produït.

La part més complicada de l'ús de l'`assert` és localitzar quins punts del codi cal comprovar i assolir un cert equilibri: massa `assert` poden convertir el codi en il·legible i ineficient; massa pocs o mal triats poden ser insuficients per depurar els nostres errors.

També s'ha de tenir en compte que el codi propi dels `assert` pot ser una font addicional d'errors: si en comptes de

```
assert(fil == m2.size());
```

feu

```
assert(fil = m2.size());
```

no estareu localitzant la situació d'error que buscàveu i, a més, donareu un valor incorrecte a la variable `fil` si les matrius tenen dimensions diferents.

Fer servir l'`assert` té un cost associat perquè s'han d'avaluar les expressions. En un programa llarg podríem arribar a posar molts `assert`. Un cop que estiguem satisfets amb el nostre codi, si volem evitar aquest cost extra, és molt fàcil deixar els `assert` sense efecte. Únicament cal afegir `#define NDEBUG` en la línia anterior a `#include <cassert>`. Això és molt més pràctic que esborrar o comentar totes les línies amb `assert`. Si més endavant voleu tornar a activar els `assert`, només haureu de comentar la línia del `#define NDEBUG`.