Herramienta para diagnósticos

La macro `assert` [1] es utilizada para incluir diagnósticos en el código con el objetivo de detectar posibles errores de programación.
La definición es:

```
void assert(int expresion);
```

Si la evaluación de *expresion* es 0 (o sea, si no es verdadera) se imprime un mensaje de error y se termina la ejecución del programa. El mensaje permite determinar donde se produjo el error.
Se incluye en el código para comprobar, por ejemplo, que se cumplan las pre y post condiciones.
Para poder usarla se debe incluir la biblioteca estándar `assert.h`.

Ejemplo:

```
// uso_assert.cpp
#include <assert.h>
#include <stdio.h>

void asigna(int * array, int n, int i, int valor) {
  assert((i >= 0) && (i < n));
  array[i] = valor;
}

int main() {
  const int TAMANIO = 10;
  int arreglo[TAMANIO];
  int pos = 10;
  asigna(arreglo, TAMANIO, pos, 100);
  printf("El valor asignado es %d.\n", arreglo[pos]);
  return 0;
}
```

El resultado de la compilación y ejecución es:

```
$ g++ uso_assert.cpp -o uso_assert
$ ./uso_assert
uso_assert: uso_assert.cpp:5: void asigna(int*, int, int, int):Assertion ‘(i >=
 0) && (i < n)’ failed.
Abortado (‘core’ generado)
```

Como la evaluación de las condiciones propuestas en las invocaciones a `assert` enlentecen la ejecución del programa, una vez que se terminó la etapa de desarrollo (cuando se tiene una razonable convicción de que se han depurado los errores) se debe proceder a remover esas invocaciones. Para lograr esto no es necesario removerlas del código, sino que incluyendo `-DNDEBUG` entre las opciones de compilación (ver `Makefile`) en la etapa de preprocesamiento se remueven de manera automática.

Precaución Una consecuencia de la remoción las invocaciones a `assert` es que se debe tener la precaución de que las expresiones que se le pasan como parámetro no tengan efectos laterales, porque esos efectos dejarán de concretarse cuando las invocaciones sean removidas del código.
Por ejemplo, supongamos que la función `es_cero_y_asigna` asigna un valor en una posición de un arreglo y además devuelve `true` si y sólo si el anterior valor en esa posición era 0. En la función que llama a `es_cero_y_asigna` se considera que es un error intentar cambiar un valor distinto de 0 y se pretende resolver esto con el uso de `assert`:

```
// assert_efectos_laterales
```

[1] Una macro es un fragmento de código al que se le da un nombre. En la etapa de preprocesamiento cada ocurrencia de ese nombre es sustituida por el código.

```
#include <assert.h>
#include <stdio.h>

bool es_cero_y_asigna(int * array, int n, int i, int valor) {
  assert((i >= 0) && (i < n));
  bool res = (array[i] == 0);
  array[i] = valor;
  return res;
}

int main() {
  const int TAMANIO = 10;
  int arreglo[TAMANIO] = {0}; // inicializa arreglo
  int pos = 9;
  assert(es_cero_y_asigna(arreglo, TAMANIO, pos, 100));
  printf("El valor asignado es %d.\n", arreglo[pos]);
  return 0;
}
```

Como `arreglo[pos]` es distinto de 0 la asignación debe hacerse:

```
$ g++ assert_efectos_laterales.cpp -o assert_efectos_laterales
$ ./assert_efectos_laterales
El valor asignado es 100.
```

Pero al remover la invocación de `assert` no se hace la asignación:

```
$ g++ -DNDEBUG assert_efectos_laterales.cpp -o assert_efectos_laterales
$ ./assert_efectos_laterales
El valor asignado es 0.
```

Lo que aqui ocurre es que en este caso la asignación es el efecto lateral. Lo que se intentaba hacer se puede resolver sustituyendo la anterior invocacion de `assert` por

```
  bool es_cero = es_cero_y_asigna(arreglo, TAMANIO, pos, 100);
  assert(es_cero);
```

En este caso la invocación de `assert` no tiene efectos laterales.