Tarea 2
Manejo dinámico de estructuras lineales
Curso 2018

Índice

1. Introducción y objetivos

En la presente tarea se trabajará sobre el manejo dinámico de memoria con estructuras lineales, en particular con nodos doblemente enlazados.
A partir de esta tarea se utilizará el analizador de uso de memoria `valgrind`. Esta herramienta debe estar instalada en las máquinas en las que se prueban los programas. Está disponible en las máqinas `Linux` de la facultad. El correcto uso de la memoria será parte de la evaluación.
Se puede trabajar en forma individual o en grupos de dos. Luego de que Bedelía entregue la lista de habilitados tras el control de previaturas, se habilitará el inscriptor de grupos.

La tarea otorga un punto a los grupos que la aprueben en la primera instancia de evaluación. La adjudicación de los puntos de las tareas de laboratorio queda condicionada a la respuesta correcta de una pregunta sobre el laboratorio en el segundo parcial. La adjudicacion se aplica de manera individual a cada estudiante del grupo.

El resto del presente documento se organiza de la siguiente forma. En la Sección 2 se presenta una descripción de los materiales disponibles para realizar la presente tarea, y en la Sección 3 se detalla el trabajo a realizar. Luego, en la Sección 4 se explica mediante ejemplos el comportamiento esperado de algunas de las funciones a implementar. La Sección 5 describe el formato y mecanismo de entrega, así como los plazos para realizar la misma. En la sección 6 se muestra un ejemplo gráfico de una cadena. En las secciones 7, 8 y 9 se presentan breves instructivos de uso de `Makefile`, `Valgrind` y `assert` respectivamente. Finalmente, en la sección 10 se sugiere un método de trabajo.

2. Materiales

Los materiales para realizar esta tarea se encuentran en el archivo *MaterialesTarea2.tar.gz* que se obtiene en la carpeta *Materiales* de la sección *Laboratorio* del sitio EVA del curso [1].
A continuación se detallan los archivos que se entregan. La estructura de directorios es igual a la de la tarea anterior.

- **info.h**: módulo de definición del tipo `info_t`
- **cadena.h**: módulo de definición de `cadena_t` y `localizador_t`.
- **uso_cadena.h**: módulo de definición de funciones sobre cadenas
- Casos de prueba en el directorio **test**
- **Makefile**: archivo para usar con el comando `make`, que provee las reglas `principal`, `testing`, `clean` como en la tarea anterior y `entrega`.
- **principal.cpp**: módulo principal.
- En el directorio *src* se entregan además:
  - los archivos *ejemplo_cadena_1_cpp.png* y *ejemplo_cadena_2_cpp.png*. Estos archivos contienen la imagen de la implementación de los tipos y algunas de las funciones que se piden,
  - el archivo *plantilla_info.cpp* con una implementación de *info.cpp*.

  El contenido de estos archivos **puede** copiarse en los archivos a implementar.

Ninguno de estos archivos deben ser modificados.

3. ¿Qué se pide?

Para cada archivo de encabezamiento **.h**, descripto en al Sección 2, se debe implementar un archivo **.cpp** que implemente **todas** las definiciones de tipo y funciones declaradas en el archivo de encabezamiento correspondiente. Los archivos **cpp** serán los entregables y deben quedar en el directorio **src**.

[1] https://eva.fing.edu.uy/course/view.php?id=132&section=6

3.1. Verificación de la implementación

Para testear la implementación debe generar el ejecutable y cumplir con todas las pruebas utilizando los archivos **.in** y **.out**. Tenga en cuenta que el archivo **Makefile** proveerá las reglas **principal**, **testing** y **clean** como en las tareas anteriores.

4. Descripción de los módulos y algunas funciones

4.1. Módulo *info*

En este módulo se declara el tipo `info_t`, como en la tarea 1. Se removió la operación `combinar_info`.

4.2. Módulo *cadena*

El módulo declara dos conceptos, denominados `cadena_t` y `localizador_t`. Con `cadena_t` se implementan cadenas doblemente enlazadas de elementos de `info_t` con una cabecera con punteros al inicio y al final. El tipo `localizador_t` es un puntero a un nodo de una cadena, o no es válido, en cuyo caso su valor es `NULL`. Con el uso de los localizadores se pueden realizar algunas operaciones de manera eficiente (sin tener que hacer un recorrido de la cadena).
A continuación se muestran gráficamente ejemplos de ejecución de algunas de las operaciones que se solicitan implementar. Cuando la ubicación de los localizadores es relevante se indica mediante colores.

4.2.1. *insertar_antes(i, loc, cad)*

i	cad (loc)	cad luego de la ejecución
(3,t)	[(1,u),(5,c),(2,d),(8,o)]	[(3,t),(1,u),(5,c),(2,d),(8,o)]
(3,t)	[(1,u),(5,c),(2,d),(8,o)]	[(1,u),(5,c),(3,t),(2,d),(8,o)]

4.2.2. *remover_de_cadena(loc, cad)*

cad (loc)	cad luego de la ejecución
[(1,u),(5,c),(2,d),(8,o)]	[(5,c),(2,d),(8,o)]
[(1,u),(5,c),(2,d),(8,o)]	[(1,u),(5,c),(2,d)]
[(1,u),(5,c),(2,d),(8,o)]	[(1,u),(5,c),(8,o)]

4.2.3. *intercambiar(loc1, loc2, cad)*

cad (loc1 y loc2)	cad luego de la ejecución
[(1,u),(3,t),(5,c),(9,n),(6,s)]	[(1,u),(9,n),(5,c),(3,t),(6,s)]

4.2.4. *siguiente_clave(clave, loc, cad)*

clave	cad (loc)	resultado (representado en cad)
8	[] (loc es ignorado)	localizador no válido
3	[(1,u),(4,c),(2,d)]	localizador no válido
11	[(1,u),(4,c),(2,d),(11,o),(9,n)]	[(1,u),(4,c),(2,d),(11,o),(9,n)]
4	[(1,u),(4,c),(2,d),(11,o),(9,n)]	[(1,u),(4,c),(2,d),(11,o),(9,n)]

4.2.5. *separar_segmento(desde, hasta, cad)*

cad (desde y hasta)	Cadena resultado	cad luego de la ejecución
[(1,u),(3,t),(5,c),(9,n),(6,s)]	[(1,u),(3,t),(5,c),(9,n)]	[(6,s)]
[(1,u),(3,t),(5,c),(9,n),(6,s)]	[(5,c),(9,n),(6,s)]	[(1,u),(3,t)]
[(1,u),(3,t),(5,c),(9,n),(6,s)]	[(3,t),(5,c),(9,n)]	[(1,u),(6,s)]

4.2.6. *mezcla(c1, c2)*

c1	c2	Cadena resultado
[]	[]	[]
[]	[(1,d),(2,b),(7,c)]	[(1,d),(2,b),(7,c)]
[(1,d),(2,b),(7,c)]	[]	[(1,d),(2,b),(7,c)]
[(1,a),(2,j)]	[(1,d),(2,b),(7,c)]	[(1,a),(1,d),(2,j),(2,b),(7,c)]

4.3. Módulo *uso_cadena*

uso_cadena contiene funciones que utilizan las estructuras definidas en el módulo descrito anteriormente. De la misma forma describiremos de forma gráfica algunas de ellas.

4.3.1. *subcadena(menor, mayor, cad)*

menor	mayor	cad	Cadena resultado
1	2	[(1,a),(2,b),(7,c)]	[(1,a),(2,b)]
3	7	[(1,a),(2,b),(3,a),(3,c),(3,b),(7,c)]	[(3,a),(3,c),(3,b),(7,c)]

5. Entrega

Se mantienen las consideraciones reglamentarias y de procedimiento de la tarea anterior.
Se debe entregar el siguiente archivo, que contiene los módulos implementados *info.cpp*, *cadena.cpp* y *uso_cadena.cpp*:

- **Entrega2.tar.gz**

Este archivo se obtiene al ejecutar la regla `entrega` del archivo *Makefile*:

```
$ make entrega
tar zcvf Entrega2.tar.gz -C src info.cpp cadena.cpp uso_cadena.cpp
info.cpp
cadena.cpp
uso_cadena.cpp
```

Con esto se empaquetan los módulos implementados y se los comprime.
Nota: En la estructura del archivo de entrega los módulos implementados deben quedar en la raíz, NO en el directorio `src` (y por ese motivo se usa la opción `-C src` en el comando `tar`).

NO SE PUEDEN ENTREGAR MÓDULOS ADICIONALES A LOS SOLICITADOS

5.1. Plazos de entrega

El plazo para la entrega es el **miércoles 11 de abril a las 14 horas**.

NO SE ACEPTARÁN ENTREGAS DE TRABAJOS FUERA DE FECHA Y HORA. LA NO ENTREGA O LA ENTREGA FUERA DE LOS PLAZOS INDICADOS IMPLICA LA PÉRDIDA DEL CURSO.

5.2. Identificación de los archivos de las entregas

Cada uno de los archivos a entrega debe contener, en la primera línea del archivo, un comentario con el número de cédula de el o los estudiantes, **sin el guión y sin dígito de verificación**. Ejemplo:

```
/* 1234567 - 2345678 */
```

6. Ejemplo de cadena

En la Figura 1 se muesta un ejemplo de cadena_t.

La variable *cad* de tipo `cadena_t` es un puntero a una estructura de dos miembros, *inicio* y *final* que son punteros a `nodo`. Un elemento de tipo `nodo` es una estructura con tres miembros: *anterior* y *siguiente* de tipo puntero a `nodo`, y *dato* de tipo `info_t`. El tipo `info_t` es un puntero a una estructura de dos miembros: *numero*, de tipo `int` (con valor 13 en el ejemplo) y *frase*, de tipo `char *` (en el ejemplo contiene el string "uy").

Las variables *loc1*, *loc2* y *loc3* son de tipo `localizador_t`, que es un puntero a `nodo`.

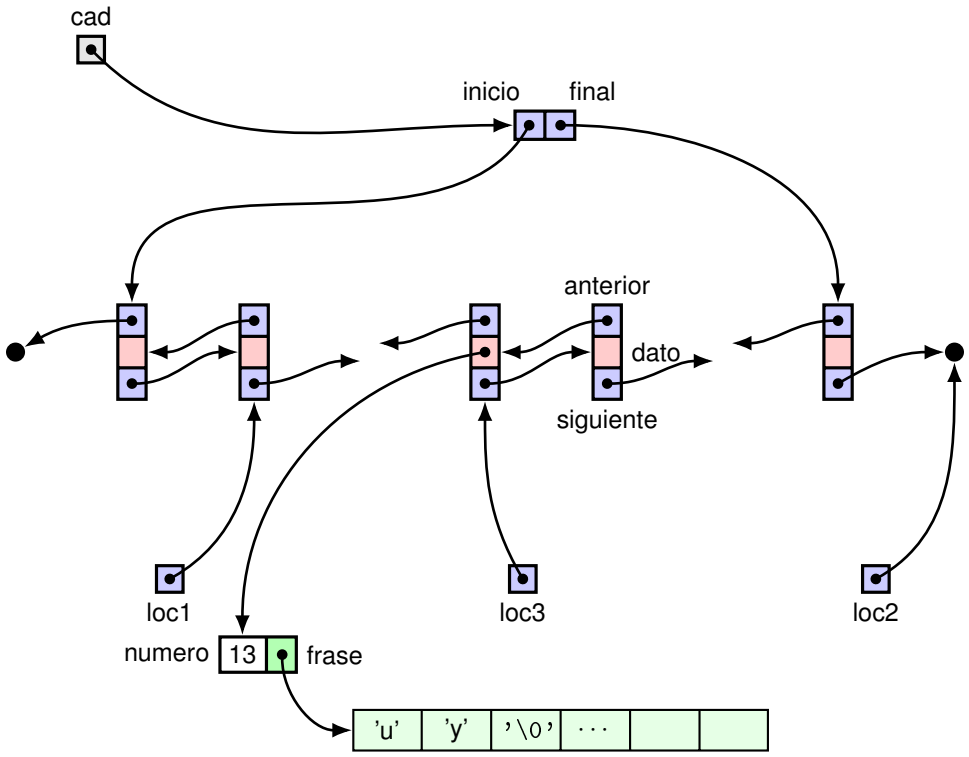

Figura 1: Ejemplo de cadena

En este ejemplo, accediendo a la representación de `cadena`, se puede afirmar que se cumplen los predicados

- `loc1 == cad->inicio->siguiente`,
- `loc2 == cad->final->siguiente`,
- `loc2 == NULL`.

Usando las operaciones de `cadena` se cumplen, entre otros:

- `! es_vacia_cadena(cad)`,
- `loc1 == siguiente(inicio_cadena(cad), cad)`,
- `loc2 == siguiente(final_cadena(cad), cad)`,
- `! es_localizador(loc2)`,
- `localizador_en_cadena(loc1, cad)`,
- `! localizador_en_cadena(loc2, cad)`,
- `precede_en_cadena(loc1, loc3, cad)`.

Nótese que no se puede expresar `loc3->dato->numero == 13` porque desde `cadena` no se puede acceder a la representación de `info_t`. En cambio es válido `numero_info(loc3->dato) == 13` o, usando las operaciones de `cadena`,
`numero_info(info_cadena(loc3, cad)) == 13`.

7. Makefile

Para automatizar el proceso de desarrollo se entrega el archivo `Makefile` que consiste en un conjunto de reglas para la utilidad `make`.
Cada regla consiste en un `objetivo`, las `acciones` para conseguir el objetivo y las `dependencias` del objetivo. Cuando el objetivo y las dependencias son archivos, las acciones se ejecutan cuando el objetivo no está actualizado respecto a las dependencias (o sea, es un archivo que no existe o su fecha de modificación es anterior a la de alguna de las dependencias). Por más información ver el manual de `make`: https://www.gnu.org/software/make/manual/.

En el *Makefile* entregado las reglas incluidas son:

- principal: para compilar y enlazar.
- clean: para borrar archivos.
- testing: para hacer pruebas.

make principal La regla `principal` compila *info.cpp* y *principal.cpp* y luego genera el ejecutable *principal*. Esta regla es la predeterminada, o sea que es la que se invoca si no se especifica ninguna.

En la siguiente secuencia se ve una ejecución exitosa de *make* junto con el estado de los directorios antes y después.

```
$ ls
include  Makefile  obj  principal.cpp  src  test
```

```
$ ls obj/
```

```
$ make
g++ -Wall -Werror -Iinclude -g -c principal.cpp -o obj/principal.o
g++ -Wall -Werror -Iinclude -g -c src/info.cpp -o obj/info.o
g++ -Wall -Werror -Iinclude -g -c src/cadena.cpp -o obj/cadena.o
g++ -Wall -Werror -Iinclude -g -c src/uso_cadena.cpp -o obj/uso_cadena.o
g++ -Wall -Werror -Iinclude -g obj/principal.o obj/info.o obj/cadena.o
obj/uso_cadena.o -o principal
```

```
$ ls
include  Makefile  obj  principal  principal.cpp  src  test
```

```
$ ls obj/
cadena.o  info.o  principal.o  uso_cadena.o
```

Si no se hacen cambios y se vuelve a correr `make` no se hace nada.

```
$ make
make: No se hace nada para «all».
```

Si hay errores de compilación se muestran en la salida estándar. En el siguiente ejemplo se muestra que la funcion `strcpy` se está invocando en la línea 100 de `info.cpp` con un argumento pero requiere dos.

```
$ make
g++ -Wall -Werror -Iinclude -g -c src/info.cpp -o obj/info.o
src/info.cpp: In function ‘rep_info* leer_info(int)’:
src/info.cpp:100:17: error: too few arguments to function ‘char* strcpy(char*, const char*)’
     strcpy(frase);
                 ^
In file included from src/info.cpp:13:0:
/usr/include/string.h:129:14: note: declared here
 extern char *strcpy (char *__restrict __dest, const char *__restrict __src)
              ^
make: *** [obj/info.o] Error 1
```

make testing Con la regla `testing` se ejecuta el programa con los casos de entrada (*.in*) generando archivos con la extensión *.sal* y estos se comparan con las salidas esperadas (*.out*), obteniendo archivos con extensión *.diff*.
Si no existe el ejecutable, se proceda a la compilación para obtenerlo. Esto es lo que se ve en el siguiente ejemplo:

```
$ make testing
g++ -Wall -Werror -Iinclude -g -c principal.cpp -o obj/principal.o
g++ -Wall -Werror -Iinclude -g -c src/info.cpp -o obj/info.o
g++ -Wall -Werror -Iinclude -g -c src/cadena.cpp -o obj/cadena.o
g++ -Wall -Werror -Iinclude -g -c src/uso_cadena.cpp -o obj/uso_cadena.o
g++ -Wall -Werror -Iinclude -g obj/principal.o obj/info.o obj/cadena.o obj/uso_cadena.o -o principal
valgrind -q --leak-check=full ./principal < test/01.in > test/01.sal 2>&1
valgrind -q --leak-check=full ./principal < test/02.in > test/02.sal 2>&1
valgrind -q --leak-check=full ./principal < test/03.in > test/03.sal 2>&1
valgrind -q --leak-check=full ./principal < test/04.in > test/04.sal 2>&1
valgrind -q --leak-check=full ./principal < test/05.in > test/05.sal 2>&1
valgrind -q --leak-check=full ./principal < test/06.in > test/06.sal 2>&1
valgrind -q --leak-check=full ./principal < test/07.in > test/07.sal 2>&1
valgrind -q --leak-check=full ./principal < test/08.in > test/08.sal 2>&1
valgrind -q --leak-check=full ./principal < test/09.in > test/09.sal 2>&1
```

Si los archivos a comparar no son iguales se indica en la salida estándar.

```
$ make testing
valgrind -q --leak-check=full ./principal < test/01.in > test/01.sal 2>&1
valgrind -q --leak-check=full ./principal < test/02.in > test/02.sal 2>&1
valgrind -q --leak-check=full ./principal < test/03.in > test/03.sal 2>&1
---- ERROR en caso test/03.diff ----
valgrind -q --leak-check=full ./principal < test/04.in > test/04.sal 2>&1
valgrind -q --leak-check=full ./principal < test/05.in > test/05.sal 2>&1
---- ERROR en caso test/05.diff ----
valgrind -q --leak-check=full ./principal < test/06.in > test/06.sal 2>&1
valgrind -q --leak-check=full ./principal < test/07.in > test/07.sal 2>&1
valgrind -q --leak-check=full ./principal < test/08.in > test/08.sal 2>&1
valgrind -q --leak-check=full ./principal < test/09.in > test/09.sal 2>&1
-- CASOS CON ERRORES --
03
05
```

Si se vuelve a correr sólo se muestran los casos con errores:

```
-- CASOS CON ERRORES --
03
05
```

Se puede ver que los archivos `.diff` de los casos con errores no tienen tamaño nulo.

```
$ stat --print="%n  %s \n" test/*.diff
test/01.diff 0
test/02.diff 0
test/03.diff 5
test/04.diff 0
test/05.diff 12
test/06.diff 0
test/07.diff 0
test/08.diff 0
test/09.diff 0
```

make clean En ocasiones puede ser útil borrar los archivos generados. Esto se hace con `make clean`. Si solo se desea borrar los archivos generados por la compilación (*info.o* y *principal*) se debe invocar `make clean_bin`. Para borrar solo los archivos generados por la ejecución de *principal* se debe invocar `make clean_test`.

8. Control de manejo de memoria

La memoria que se obtiene dinámicamente (mediante `new`) debe ser liberada (con `delete`). Esto debe hacerse de manera sistemática: por cada instrucción que solicita memoria debe haber alguna o algunas correspondientes que la liberen. No obstante, además de esta buena práctica de programación, se debe controlar que el programa hace un correcto manejo de le memoria. Una vez que se sabe que el programa cumple la especificación funcional se debe correr alguna herramienta de análisis. Un ejemplo de ello es `valgrind`.

El siguiente es el resultado de la ejecución de un programa que deja sin liberar 1 bloque de 16 bytes:

```
$ valgrind ./principal < test/01.in
==26226== Memcheck, a memory error detector
==26226== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26226== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26226== Command: ./principal
==26226==
1>#  prueba crear_info y liberar_info.
2>Fin.
==26226==
==26226== HEAP SUMMARY:
==26226==     in use at exit: 16 bytes in 1 blocks
==26226==   total heap usage: 5 allocs, 4 frees, 1,122,321 bytes allocated
==26226==
==26226== LEAK SUMMARY:
==26226==    definitely lost: 16 bytes in 1 blocks
==26226==    indirectly lost: 0 bytes in 0 blocks
==26226==      possibly lost: 0 bytes in 0 blocks
==26226==    still reachable: 0 bytes in 0 blocks
==26226==         suppressed: 0 bytes in 0 blocks
==26226== Rerun with --leak-check=full to see details of leaked memory
==26226==
==26226== For counts of detected and suppressed errors, rerun with: -v
==26226== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Al comando de ejecución se le puede agregar la opción `--leak-check=full` para obtener más informacion:

```
$ valgrind --leak-check=full ./principal < test/01.in
==26400== Memcheck, a memory error detector
==26400== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26400== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26400== Command: ./principal
==26400==
1>#  prueba crear_info y liberar_info.
2>Fin.
==26400==
==26400== HEAP SUMMARY:
==26400==     in use at exit: 16 bytes in 1 blocks
```

```
==26400==   total heap usage: 5 allocs, 4 frees, 1,122,321 bytes allocated
==26400==
==26400== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==26400==    at 0x4C2E1FC: operator new(unsigned long) (vg_replace_malloc.c:334)
==26400==    by 0x400F97: crear_info(int, char*) (info.cpp:23)
==26400==    by 0x400A06: main (principal.cpp:47)
==26400==
==26400== LEAK SUMMARY:
==26400==    definitely lost: 16 bytes in 1 blocks
==26400==    indirectly lost: 0 bytes in 0 blocks
==26400==      possibly lost: 0 bytes in 0 blocks
==26400==    still reachable: 0 bytes in 0 blocks
==26400==         suppressed: 0 bytes in 0 blocks
==26400==
==26400== For counts of detected and suppressed errors, rerun with: -v
==26400== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Se muestra que en la línea 47 de `principal.cpp` se llama a `crear_info`, que en la línea 23 de `info.cpp` usa el operador `new` para obtener memoria. Ese segmento de memoria obtenido es el que no fue liberado. Luego de agregar el `delete` en la función que debe devolver ese bloque se puede ver que no hay errores ni memoria perdida.

```
$ valgrind ./principal < test/01.in
==26905== Memcheck, a memory error detector
==26905== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26905== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==26905== Command: ./principal
==26905==
1>#  prueba crear_info y liberar_info.
2>Fin.
==26905==
==26905== HEAP SUMMARY:
==26905==     in use at exit: 0 bytes in 0 blocks
==26905==   total heap usage: 5 allocs, 5 frees, 1,122,321 bytes allocated
==26905==
==26905== All heap blocks were freed -- no leaks are possible
==26905==
==26905== For counts of detected and suppressed errors, rerun with: -v
==26905== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

---

9. assert

La macro `assert` [2] 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.

[2] 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.

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
#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.

10. Método de trabajo sugerido

Como método de trabajo se sugiere que la tarea se desarrolle en una iteración en la que en cada paso se implementa, prueba e integra una nueva funcion. Una forma alternativa consiste en primero implementar todo y luego probarlo. Con este método es más dificil determinar dónde están los errores y una vez encontrados puede requerir muchos cambios en el resto del programa. Con el método iterativo, en cambio, es probable que los errores sean detectados rápidamente y que no impacten en el resto del código.

Se sugiere que antes de empezar a implementar se estudie toda la especificación y se determine el orden en el que se va a implementar y probar cada tipo y función. Este orden debe respetar dependencias. Por ejemplo no se puede intentar probar una función que remueve elementos de una colección si antes no se implementó el tipo, una función que cree la colección y otra que inserte elementos en ella.

Para proceder con este método se necesitan herramientas con las cuales realizar las pruebas. Este es la objetivo del módulo `principal.cpp`. Consiste en un procesador de comandos, con un comando para cada función a implementar. Cuando se va a implementar una función, por ejemplo `insertar_al_final` se busca el segmento de código de `principal.cpp` en donde se la prueba:

```
} else if (!strcmp(nom_comando, "insertar_al_final")) {
  info_t info = leer_info(MAX_LINEA);
  insertar_al_final(info, cad);
  printf("Insertado al final.\n");
```

para determinar cuales son el formato y el comportamiento esperados. Lo mismo puede deducirse observando los archivos `.in` y `.out` que se suministran en el archivo de materiales.

Entonces, lo que se debe hacer es crear un archivo de texto que será la entrada del programa, por ejemplo `test/prueba.in`, en el que de manera incremental se irán agregando comandos que prueban las funciones, y otro, `test/prueba.out`, en el que se escribe el resultado esperado. Cada vez que se implementa una función se compila, se corre el programa y se comprueba que la salida es correcta.
El archivo de entrada más básico y aún así útil es el que sólo tiene el comando `Fin`. El programa principal crea una cadena vacía entra a un ciclo del que sale cuando se lee dicho comando y al terminar el ciclo libera la memoria de la cadena creada. Por lo tanto está probando la implementación del tipo, y las funciones que crean y liberan. Es posible que en una etapa tan temprana ya haya errores, en particular los relacionados con el manejo de memoria. Por ese motivo las corridas para probar el programa deben incluir el uso de `valgrind` (ver sección 8). El siguiente paso podría ser probar la función `es_vacia_cadena`, y en el siguiente probar `insertar_al_final`. En esta etapa el contenido de `test/prueba.in` podria ser:

```
es_vacia_cadena
insertar_al_final (1,a)
es_vacia_cadena
Fin
```

al que le corresponde `test/prueba.out`:

```
1>cad vacia.
2>Insertado al final.
3>cad no vacia.
4>Fin.
```

Después de implementar las funciones mencionadas se compila y se corre:

```
$ make
$ valgrind -q --leak-check=full ./principal < test/prueba.in
```

Si la salida no es igual al del archivo `.out` o hay errores o pérdidas de memoria se itera hasta que el resultado sea correcto. Para que la comprobación sea más rigurosa se usa la utiidad `diff`:

```
$ ./principal < test/prueba.in > test/prueba.sal
$ diff test/prueba.out test/prueba.sal
```

Y luego se pasa a la siguiente función. Cuando se está razonablemente seguro de que el programa es correcto se utiliza la regla `testing` del `Makefile` (ver seccion 7). Si es necesario se repite el proceso. En el archivo `.in` se puede poner el comando `Fin` en cualquier línea, no necesariamente la última, si sólo se quiere probar algunas funciones. Por ejemplo, con

```
es_vacia_cadena
Fin
insertar_al_final (1,a)
es_vacia_cadena
Fin
```

no se prueba `insertar_al_final`, sino sólo hasta `es_vacia_cadena`.

Finalmente se debe probar en las máquinas de la facultad, ya sea de manera presencial o remota (ver Instructivo de ambiente).