Tarea 1
Introducción a C
Curso 2018

Índice

1. Introducción y objetivos **2**

2. Materiales **2**

3. ¿Qué se pide? **3**
3.1. Verificación de la implementación . . . 3

4. Múltiples archivos **3**

5. Acerca de C **4**
5.1. Compilador . . . 4
5.2. Lenguaje C* . . . 4
5.3. Bibliotecas . . . 4

6. Entrega **4**
6.1. Plazos de entrega . . . 4
6.2. Forma de entrega . . . 5
6.3. Identificación de los archivos de las entregas . . . 5
6.4. Evaluación . . . 5
6.5. Nuevos envíos . . . 6
6.6. Segunda instancia ante entregas insuficientes . . . 6
6.7. Individualidad . . . 6
6.8. Nueva información y comunicación . . . 6
6.8.1. Uso de los foros . . . 7

7. Apéndice: información adicional **8**
7.1. Makefile . . . 8
7.2. Control de manejo de memoria . . . 10
7.3. assert . . . 11

1. Introducción y objetivos

Este laboratorio tiene los siguientes objetivos específicos:

- Realizar un primer acercamiento al lenguaje C y a algunas de sus bibliotecas básicas, en particular las que permiten procesar fragmentos de texto y acceder a la entrada y salida estándar.
- Comenzar a familiarizarse con la división del código que compone un programa en módulos distribuidos en múltiples archivos.
- Implementar el tipo `info_t` que modelará un tipo genérico durante el resto de las tareas.

En esta tarea se puede trabajar únicamente en forma individual. La entrega es **obligatoria**.

2. Materiales

Los materiales para realizar esta tarea se encuentran en el archivo *MaterialesTarea1.tar.gz* que se obtiene en la carpeta *Materiales* de la sección *Laboratorio* del sitio EVA del curso [1].
Los archivos que están en *MaterialesTarea1.tar.gz* están dispuestos en una estructura de directorios, la cual debe conservarse.

- En la raíz deben estar el módulo principal (*principal.cpp*), el *Makefile* y el ejecutable que se generará tras la compilación.
- Los archivos de encabezamiento (*.h*) están en el directorio *include* (en esta tarea es el archivo: **info.h**).
- Los archivos a implementar **deben** crearse en el directorio *src* (en esta tarea es el archivo: **info.cpp**).
- Los archivos que resultan de la compilación de cada módulo, (*.o*), se mantienen en el directorio *obj*.
- Los casos de prueba están en el directorio *test*.
- En el directorio *src* se entrega además el archivo *ejemplo_info_cpp.png*. Este archivo contiene la imagen de la implementación de los tipos y algunas de las funciones que se piden.

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

Para desempaquetar el material se usa el comando *tar*:

```
$ tar zxvf MaterialesTarea1.tar.gz
tarea1/Makefile
tarea1/principal.cpp
tarea1/include/info.h
tarea1/src/ejemplo_info_cpp.png
tarea1/obj/.ignore
tarea1/test/01.in
tarea1/test/01.out
tarea1/test/02.in
tarea1/test/02.out
tarea1/test/03.in
tarea1/test/03.out
tarea1/test/04.in
tarea1/test/04.out
tarea1/test/05.in
tarea1/test/05.out
tarea1/test/06.in
tarea1/test/06.out
tarea1/test/07.in
tarea1/test/07.out
tarea1/test/08.in
tarea1/test/08.out
tarea1/test/09.in
tarea1/test/09.out
```

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

Ninguno de estos archivos deben ser modificados.

3. ¿Qué se pide?

Para cada archivo de encabezamiento **.h**, descrito 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. Para hacerlo se **puede** usar el código incluido en los archivos *png*. Los archivos **cpp** serán los entregables y deben quedar en el directorio **src**.

3.1. Verificación de la implementación

La implementación de código es solo una parte del desarrollo de software. Tras la implementación de cada módulo, de cada función, se debe verificar que se está cumpliendo con los requerimientos.

Para testear su implementación debe generar el ejecutable (**principal**) que funcionará como verificador del módulo *info*. El mismo lee desde la entrada estándar, procesa lo leído utilizando las funciones y procedimientos que se desarrollaron en *info* y escribe en la salida estándar. Se incluyen como ejemplo algunos casos de prueba (que **NO** necesariamente serán los utilizados para la corrección). En ellos cada archivo *.in* representa lo leído desde la entrada estándar y el correspondiente archivo *.out* lo que se debe escribir en la salida estándar. Se pueden redireccionar la entrada y salida estándar con los operadores < y > respectivamente. Por lo tanto se puede ejecutar *principal* con cada archivo con extensión *.in* redirigiendo la salida hacia otro archivo (por ejemplo, con extensión *.sal*) que luego se compara con el archivo con extensión `.out` correspondiente. La comparación se hace con la utilidad *diff*. Si los archivos comaparados son iguales no se imprime nada en la salida, por lo que si la salida de la ejecución de *diff* se redirige hacia un archivo, este tendrá tamaño 0. Se muestra un ejemplo de ejecución y comparación exitosa:

```
$ ./principal < test/01.in > test/01.sal
$ diff test/01.out test/01.sal > test/01.diff
```

La generación del ejecutable se hace compilando el archivo implementado **info.cpp** y el archivo provisto en la carpeta de materiales **principal.cpp**.
El archivo **Makefile** (ver Sección 7.1) provee las reglas **principal** para la compilación y **testing** para la ejecución y comparación.

4. Múltiples archivos

Cuando la dimensión o complejidad del programa a construir crece es conveniente dividir el código en varios archivos, en cada uno de los cuales se incluyen entidades (tipos, constantes, funciones) altamente relacionadas entre sí. A estos agrupamientos se les suele denominar *módulos*. Con esta división es posible que el trabajo sea realizado de manera independiente en cada uno de los módulos, posiblemente por distintos desarrolladores. Además, algunos de esos módulos pueden reusarse para otros proyectos, o sea, servir como bibliotecas.

La división en módulos requiere que se pueda cumplir con la compilación separada, esto es, que cada uno de los módulos pueda ser compilado sin disponer del código de los otros. Para esto se presenta un problema cuando un módulo depende de entidades que serán implementadas en otros, ya que deben incluirse para poder compilar pero no se dispone de ellas. Este se soluciona separando cada módulo en dos archivos, en uno de los cuales se declaran las entidades y en el otro se las implementa. En el lenguaje `C` a los primeros se los suele identificar con la extensión *.h* (o también *.hpp* en `C++`) (la letra proviene de *headers*, encabezamientos).
En esta tarea el objetivo principal es implementar el tipo `info_t`. Este tipo se usará en el resto de las tareas para modelar un tipo general. Consiste en un dato numérico y un dato de texto.
En la actual tarea el módulo *principal* usa tipos y funciones declaradas en el módulos *info*. Para poder desarrollarlo y compilarlo se incluyen los archivos de encabezamiento.

```
#include "include/info.h"
```

Con esto, se puede compilar y obtener *principal.o* aunque no se disponga de *info.cpp*:

```
$ g++ -Wall -Werror -Iinclude -c principal.cpp -o principal.o
```

5. Acerca de C

5.1. Compilador

Todos los módulos entregados deben compilar y enlazar con el compilador del curso (g++), sin extensiones. Para la instalación del compilador, se debe ejecutar el siguiente comando:

En distribuciones que utilicen manejador APT (por ejemplo, Ubuntu)

```
sudo apt-get install g++
```

En distribuciones que utilicen manejador YUM (por ejemplo, Fedora)

```
sudo yum install gcc-c++
```

En versiones más nuevas de Fedora se utiliza el manejador DNF, por lo que el comando de instalación es:

```
sudo dnf install gcc-c++
```

5.2. Lenguaje C*

Se utilizará el lenguaje C* que es una extensión de C con algunas funcionalidades de C++. Las funcionalidades que se agregan son:

- Operadores `new` y `delete`.
- Pasaje por referencia.
- Tipo `bool`.
- Definición de `struct` y `enum` al estilo C++.

Se puede consultar las referencias de C y C++ que se encuentran en la sección Bibliografía del sitio EVA del curso.

5.3. Bibliotecas

Se espera que se utilicen las bibliotecas de entrada y salida y de manejo de *strings* de C. Como referencia de las mismas se puede verificar los siguientes enlaces:

- *stdio.hpp*: http://www.cplusplus.com/reference/cstdio/
- *stdlib.hpp*: http://www.cplusplus.com/reference/cstdlib/
- *string.hpp*: http://www.cplusplus.com/reference/cstring/

6. Entrega

Se debe entregar el archivo *info.cpp*.

6.1. Plazos de entrega

El plazo para la entrega es el **14 de marzo a las 14:00**.

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.

6.2. Forma de entrega

Las entregas se realizarán a través de la plataforma EVA del curso. Para ello se deberá acceder a la actividad que se habilitará previo a la fecha de entrega para esos fines. El archivo a entregar **DEBE** llamarse *info.cpp*.

Es importante señalar que para realizar las entregas es necesario que se encuentre matriculado al curso EVA.

Se recuerda que en los salones 314, 401 y 402 se dispone de computadoras conectadas a la red. Esto permite que se pueda acceder desde ellas al sitio EVA para realizar las entregas.

6.3. Identificación de los archivos de las entregas

Cada uno de los archivos a entregar debe contener, en la primera línea del archivo, un comentario con el número de cédula del estudiante, sin el guión y sin dígito de verificación.

Ejemplo:

```
/* 1234567 */
```

6.4. Evaluación

La tarea se evaluará con un conjunto de casos de prueba que puede incluir algunos de los publicados. En todos los casos el programa deberá funcionar correctamente de acuerdo a la especificación proporcionada.

La tarea otorgará un punto sólo si la entrega se apruebe en la primera instancia de evaluación (ver sección 6.6). Dicho punto sirve para la aprobación o exoneración del curso. La adjudicación de ese punto queda condicionada a la respuesta correcta de una pregunta sobre el laboratorio en el segundo parcial.

Se recomienda que los programas sean probados en máquinas con Linux, que es el sistema operativo en el que se realizará la corrección. En particular se debe correr en las condiciones que se especifican en el Instructivo de ambiente, que se encuentra en la sección *Lenguaje y herramientas* de *Laboratorio*. Se puede probar la implementación tanto estando de forma física en las salas como de forma remota conectándose vía SSH. El procedimiento para realizar lo segundo puede también ser consultado en ese documento.

Se debe tener presente que **las entregas que no compilen o cuya salida no sea idéntica a la de los ejemplos presentados, serán consideradas insatisfactorias**.

Además de la corrección mediante casos de prueba se podrá hacer inspección de código. Se evaluará positivamente:

- Estructuración del código.
- Calidad de los comentarios.
- Claridad en el código.
- Uso de estructuras de datos adecuadas al problema.
- Nombre de variables, constantes y procedimientos descriptivos de la función que desempeñan en el programa.

6.5. Nuevos envíos

Desde el día que se habilite la actividad asociada a la tarea y hasta la fecha de culminación, podrá realizar todos los envíos que considere necesario. Tenga en cuenta que sólo el último será corregido.

Se recomienda realizar un primer envío de prueba de forma de verificar que puede realizar el procedimiento con normalidad.

6.6. Segunda instancia ante entregas insuficientes

Finalizado el plazo de entrega se procederá a la corrección de los programas. A aquellos cuyas entregas no resulten satisfactorias se les permitirá hacer una reentrega con modificaciones de la entrega original. Esas modificaciones deben corregir pequeños errores detectados al probar el programa con nuevos casos. El plazo para la reentrega será de **24 horas desde que se publiquen los resultados y los casos de prueba usados en la corrección**. También se realizará a través del sitio EVA.

Los criterios para considerar satisfactoria una segunda instancia serán más rigurosos que los usados para la entrega original.

El control de individualidad, tanto para las entregas consideradas satisfactorias en la primera instancia como en la segunda, se realizará después de terminado el plazo de la segunda instancia.

6.7. Individualidad

Para las tareas de laboratorio vale lo establecido en el Reglamento sobre No individualidad en instancias de evaluación de la Facultad de Ingeniería.
No existirán instancias en el curso para reclamos frente a la detección por parte de los docentes de trabajos de laboratorio no individuales, independientemente de las causas que pudiesen originar la no individualidad. A modo de ejemplo y sin ser exhaustivos:

- Utilización de código realizado en cursos anteriores u otros cursos.
- Perder el código.
- Olvidarse del código en lugares accesibles a otros estudiantes.
- Prestar el código o dejar que el mismo sea copiado por otros estudiantes.
- Dejar la terminal con el usuario abierto al retirarse.
- Enviarse código por mail.
- Etc.

Es decir, se considera a cada estudiante **RESPONSABLE DE SU TRABAJO DE LABORATORIO Y DE QUE EL MISMO SEA INDIVIDUAL**.

Los trabajos de laboratorio que a juicio de los docentes no sean individuales **serán eliminados con la consiguiente pérdida del curso para todos los involucrados**. Además, todos los casos serán enviados a los órganos competentes de la Facultad, lo cual puede acarrear sanciones de otro carácter y gravedad para los estudiantes involucrados.

Asimismo **se prohibe el envío de código a los foros del curso** dado que el mismo será considerado como una forma de compartir código.

6.8. Nueva información y comunicación

En caso de ser necesario se publicarán documentos con los agregados y/o correcciones al laboratorio que puedan surgir con el avance del curso.

Toda publicación de nueva información se realizará en la sección *Laboratorio* y será notificada en las clases teóricas y prácticas. Los estudiantes cuentan con foros en el sitio EVA para discutir acerca del laboratorio.

Las casillas personales de los docentes no son el medio de comunicación adecuado para este tipo de discusiones. Por ese motivo NO se responderán consultas por este medio.

6.8.1. Uso de los foros

Es obligación leer el Reglamento de Participación en los Foros disponible en la sección *Información General del Curso* del sitio EVA.

Antes de publicar un mensaje **verifique si hay un hilo de conversación para lo que quiera consultar o comunicar**. Sólo inicie una nueva consulta en el foro en caso que no existiera un hilo que ya lo hiciera.

7. Apéndice: información adicional

7.1. 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 obj/principal.o obj/info.o -o principal

$ ls
include  Makefile  obj  principal  principal.cpp  src  test

$ ls obj/
info.o  principal.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 obj/principal.o obj/info.o -o principal
./principal < test/01.in > test/01.sal
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
./principal < test/04.in > test/04.sal
./principal < test/05.in > test/05.sal
./principal < test/06.in > test/06.sal
./principal < test/07.in > test/07.sal
./principal < test/08.in > test/08.sal
./principal < test/09.in > test/09.sal
```

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

```
$ make testing
./principal < test/01.in > test/01.sal
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
---- ERROR en caso test/03.diff ----
./principal < test/04.in > test/04.sal
./principal < test/05.in > test/05.sal
---- ERROR en caso test/05.diff ----
./principal < test/06.in > test/06.sal
./principal < test/07.in > test/07.sal
./principal < test/08.in > test/08.sal
./principal < test/09.in > test/09.sal
-- 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`.

7.2. 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
==22189== Memcheck, a memory error detector
==22189== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==22189== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==22189== Command: ./principal
==22189==
1>#  prueba crear_info y liberar_info.
2>Fin.
==22189==
==22189== HEAP SUMMARY:
==22189==     in use at exit: 16 bytes in 1 blocks
==22189==   total heap usage: 5 allocs, 4 frees, 77,841 bytes allocated
==22189==
==22189== LEAK SUMMARY:
==22189==    definitely lost: 16 bytes in 1 blocks
==22189==    indirectly lost: 0 bytes in 0 blocks
==22189==      possibly lost: 0 bytes in 0 blocks
==22189==    still reachable: 0 bytes in 0 blocks
==22189==         suppressed: 0 bytes in 0 blocks
==22189== Rerun with --leak-check=full to see details of leaked memory
==22189==
==22189== For counts of detected and suppressed errors, rerun with: -v
==22189== 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
==22320== Memcheck, a memory error detector
==22320== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==22320== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==22320== Command: ./principal
==22320==
1>#  prueba crear_info y liberar_info.
2>Fin.
==22320==
==22320== HEAP SUMMARY:
==22320==     in use at exit: 16 bytes in 1 blocks
==22320==   total heap usage: 5 allocs, 4 frees, 77,841 bytes allocated
==22320==
==22320== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==22320==    at 0x4C2E1FC: operator new(unsigned long) (vg_replace_malloc.c:334)
==22320==    by 0x400F97: crear_info(int, char*) (info.cpp:23)
==22320==    by 0x400A06: main (principal.cpp:24)
==22320==
==22320== LEAK SUMMARY:
```

```
==22320==    definitely lost: 16 bytes in 1 blocks
==22320==    indirectly lost: 0 bytes in 0 blocks
==22320==      possibly lost: 0 bytes in 0 blocks
==22320==    still reachable: 0 bytes in 0 blocks
==22320==         suppressed: 0 bytes in 0 blocks
==22320==
==22320== For counts of detected and suppressed errors, rerun with: -v
==22320== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Se muestra que en la línea 24 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
==22592== Memcheck, a memory error detector
==22592== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==22592== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==22592== Command: ./principal
==22592==
1>#  prueba crear_info y liberar_info.
2>Fin.
==22592==
==22592== HEAP SUMMARY:
==22592==     in use at exit: 16 bytes in 1 blocks
==22592==   total heap usage: 5 allocs, 4 frees, 77,841 bytes allocated
==22592==
==22592== LEAK SUMMARY:
==22592==    definitely lost: 16 bytes in 1 blocks
==22592==    indirectly lost: 0 bytes in 0 blocks
==22592==      possibly lost: 0 bytes in 0 blocks
==22592==    still reachable: 0 bytes in 0 blocks
==22592==         suppressed: 0 bytes in 0 blocks
==22592== Rerun with --leak-check=full to see details of leaked memory
==22592==
==22592== For counts of detected and suppressed errors, rerun with: -v
==22592== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

7.3. 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.
Para poder usarla se debe incluir la biblioteca estándar `assert.h`.

Ejemplo:

[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.

```
// 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.