Tarea 1
Introducción a los Tipos Abstractos de Datos
Curso 2019

Índice

1. Introducción y objetivos

Esta tarea 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.
- Conocer los TADs `Pila` y `Mapping`.
- Implementar funciones que usen los TADs `Pila` y `Mapping`.

En esta tarea se puede trabajar únicamente en forma individual. La tares es **eliminatoria**.

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 son los archivos: **pila.h**, **mapping.h** y **uso_tads.h**).
- Los archivos a implementar **deben** crearse en el directorio *src* (en esta tarea son los archivo: **pila.cpp**, **mapping.cpp** y **uso_tads.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 archivos *ejemplo_*_cpp.png*. Estos archivos 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/pila.h
tarea1/include/mapping.h
tarea1/include/uso_tads.h
tarea1/src/pila.cpp
tarea1/src/mapping.cpp
tarea1/src/ejemplo-uso_tads.cpp.png
tarea1/obj/.ignore
tarea1/test/00.in
tarea1/test/00.out
tarea1/test/01.in
tarea1/test/01.out
tarea1/test/02.in
tarea1/test/02.out
tarea1/test/03.in
tarea1/test/03.out
```

Ninguno de estos archivos deben ser modificados.

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

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**.
En esta tarea solo se debe implementar el archivo `uso_tads.cpp` ya que los arhivos `pila.cpp` y `mapping.cpp` son parte de la distribución.

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 de los módulos *pila*, *mapping* y *uso_tads*. El mismo lee desde la entrada estándar, procesa lo leído utilizando las funciones y procedimientos que se desarrollaron en *pila*, *mapping* y *uso_tads* 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 comparados 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 los archivos implementados **pila.cpp**, **mapping.cpp** y **uso_tads.cpp** y el archivo provisto en la carpeta de materiales **principal.cpp** y enlazando los archivos objeto obtenidos:

```
$ g++ -Wall -Werror -Iinclude -g -c principal.cpp -o obj/principal.o
$ g++ -Wall -Werror -Iinclude -g -c src/pila.cpp -o obj/pila.o
$ g++ -Wall -Werror -Iinclude -g -c src/mapping.cpp -o obj/mapping.o
$ g++ -Wall -Werror -Iinclude -g -c src/uso_tads.cpp -o obj/uso_tads.o
$ g++ -Wall -Werror -Iinclude -g obj/principal.o obj/pila.o
  obj/mapping.o obj/uso_tads.o -o principal
```

El archivo **Makefile** (ver Sección 7) provee las reglas **principal** para la compilación y **testing** para la ejecución y comparación.

4. Descripción de los módulos

En esta sección se desciben los módulos que componen la tarea. Estos son `pila`, `mapping`, `uso_tads` y `principal`.
Los módulos `pila` y `mapping` son TADs y el módulo `uso_tads` consiste en funciones para cuya implementación puede ser útil usar los TADs anteriores.
El módulo `principal` es un intérprete de comandos utilizado para probar las funciones implementadas en los otros módulos. Consiste en un único archivo, `principal.cpp`. Para poder compilar este archivo de manera separada, es decir, sin tener aún las implementaciones de los otros módulos, cada uno de estos últimos se dividen en dos: especificación (`.h`) e implementación (`.cpp`).

En los archivos de especificación se declaran constantes, tipos y funciones. Entonces si en `principal.cpp` se incluyen estos archivos mediante

```
#include "include/pila.h"
#include "include/mapping.h"
#include "include/uso_tads.h"
```

se pueden usar las entidades ahí declaradas y el compilador puede comprobar que el código fuente de `principal.cpp` es sintácticamente correcto y entonces generar el código objeto correspondiente.
De la misma forma si en `uso_tads.cpp` se incluyen los archivos de especificación de los TADs se puede implementar las funciones y compilar de manera separada.

En los archivos de especificación de los TADs se declaran tipos que serán implementados en los archivos de implementación correspondientes. Por ejemplo en `pila.h` se define el tipo `pila_t`:

```
typedef struct rep_pila *pila_t;
```

Con esto se sabe que los elementos de tipo `pila_t` son punteros y por lo tanto una variable de este tipo ocupa 4 bytes. De esta manera si en `principal.cpp` o en `uso_tads.cpp` se declara una variable de este tipo el compilador conoce cuánto espacio va a ocupar, lo que permite generar el código objeto.
El tipo `struct rep_pila` (la representación de `pila_t`) es el tipo de los elementos a los que apuntan los elementes de tipo `pila_t`. Este tipo queda declarado pero su definición debe implementarse en el archivo `pila.cpp`. El ámbito de esa definición será solo el archivo `pila.cpp`. Un intento de declarar una variable de tipo `struct rep_pila` en `principal.cpp` o en `uso_tads.cpp` generará un error de compilación.

En lo que sigue se detallan los módulos `pila`, `mapping` y `uso_tads`.

4.1. Módulo *pila*

El TAD `Pila` modela las secuencias de elementos en las que el último elemento ingresado es el único al que se tiene acceso. En esta tarea los elementos son de tipo `char`.
Las operaciones estándar del TAD `Pila` son:

Crear que devuelve una pila sin elementos.

Apilar que agrega un elemento a la pila. O sea, si la pila consistía en la secuencia $(e_1, e_2, \cdots, e_n)$ y se agrega el elemento e el resultado consiste en la secuencia $(e_1, e_2, \cdots, e_n, e)$.

Es_vacia que indica si la pila tiene elementos o no.

Tope que devuelve el último elemento agregado en la pila, que no puede ser vacía. O sea, si la pila consiste en la secuencia $(e_1, e_2, \cdots, e_n)$, con $n > 0$, devuelve el elemento e_n.

Desapilar que remueve de la pila, que no puede ser vacía, el último elemento agregado en la pila. O sea, si la pila consistía en la secuencia $(e_1, e_2, \cdots, e_{n-1}, e_n)$, con $n > 0$, el resultado consiste en la secuencia $(e_1, e_2, \cdots, e_{n-1})$.

En esta tarea se establece la restricción de que la pila debe ser acotada, o sea que la cantidad máxima de elementos que puede contener está determinada por una constante. Esto hace que el TAD se extienda con otra operación:

Es_llena que indica si la cantidad de elementos de la pila es igual al máximo permitido.

Como consecuencia, la operacion `Apilar` solo está definida si la pila no está llena.

4.2. Módulo *mapping*

El TAD `mapping` modela un conjunto de pares ordenados $(clave, valor)$ tal que el primer componente no se repite. O sea un mapping es un conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$, con $k_i \neq k_j$ si $i \neq j$. De esta manera cada clave del mapping tiene asociado un valor. Se debe notar que en general un mismo valor puede estar asociado a más de una clave. El conjunto al que pertenecen las claves y el conjunto al que pertenecen los valores asociados pueden ser el mismo o no. Por ejemplo las claves pueden ser ciudades y los valores asociados la cantidad de habitantes. En esta tarea tanto las claves como los valores son caracteres, es decir, elementos de tipo `char`.
Las operaciones estándar del TAD `Mapping` son:

Crear que devuelve en conjunto vacío, sin pares.

Asociar que asocia una clave k, que no pertenecía al conjunto de claves del mapping, con un valor v. La clave y el valor dados constituyen un nuevo par del conjunto de asociaciones del mapping. O sea, si el mapping consistía en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$ y se asocia el par (k, v), con $k \neq k_i$ para $1 \leq i \leq n$, el resultado consiste en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n), (k, v)\}$.

Es_clave que indica si una clave k pertenece al conjunto de claves del mapping. O sea, si el mapping consiste en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$ la clave k es clave del mapping si y solo si pertenece al conjunto $\{k_1, \cdots, k_n\}$.

Valor que devuelve el valor asociado a una clave k, que es clave del mapping. O sea, si el mapping consiste en el conjunto $\{(k_1, v_1), \cdots, (k_i, v_i), \cdots, (k_n, v_n)\}$ y $k = k_i$ el valor asociado a k es v_i.

Desasociar que remueve del mapping la asociación correspondiente a una clave k que es clave del mapping. O sea, si el mapping consistía en el conjunto $\{(k_1, v_1), \cdots, (k_{i-1}, v_{i-1}), (k_i, v_i), (k_{i+1}, v_{i+1}), \cdots, (k_n, v_n)\}$ y $k = k_i$ el resultado consiste en el conjunto $\{(k_1, v_1), \cdots, (k_{i-1}, v_{i-1}), (k_{i+1}, v_{i+1}), \cdots, (k_n, v_n)\}$.

En esta tarea se establece la restricción de que el mapping debe ser acotado, o sea que la cantidad máxima de pares que puede contener está determinada por una constante. Además se quiere poder determinar si un determinado valor está asociado a alguna clave.
Esto hace que el TAD se extienda las operaciones:

Es_lleno que indica si la cantidad de elementos del mapping es igual al máximo permitido.

Es_valor que indica si un valor v está asociado a alguna clave del mapping. O sea, si el mapping consiste en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$ el valor v es valor del mapping si y solo si existe i tal que $v = v_i$.

La operacion `Asociar` solo está definida si el mapping no está lleno.

4.3. Módulo *uso_tads*

Este módulo consiste en funciones para cuya implementación puede ser útil usar los TADs antreriores. En esta instancia hay una única funcion:

```
#define MAX_EXP 40

/*
  Devuelve 'true' si y solo si los delimitadores de 'expresion' están
  correctamente balanceaados.
  Los delimitadores de apertura son
   '<', '(', '{' y '[' y
  los correspondientes  delimitadores de cierre son
   '>', ')', '}' y ']'.
  Los demás caracteres no son considerados delimitadores.
  Los caracteres están desde la posición 0 hasta la n - 1 de
  'expresion'.
 */
bool es_balanceado(char expresion[MAX_EXP], int n);
```

Un string es balanceado si sus delimitadores cumplen las reglas habituales de las expresiones matemáticas:

- Por cada delimitador de apertura que ocurre debe aparecer luego su correspondiente delimitador de cierre. No está balanceado el string "(".
- Por cada delimitador de cierre que ocurre debe aparecer antes su correspondiente delimitador de apertura. No está balanceado el string ">".
- Dos pares de delimitadores no se pueden solapar. Llamemos rango de un par de delimitadores correspondientes al conjunto de posiciones que van desde el delimitador de apertura al delimitador de cierre. Entonces dados dos pares de delimitadores o bien sus rangos son disjuntos o bien uno de los rangos está contenido dentro del otro. Por ejemplo los strings "()<>" y "[<>]" están balanceados. En cambio el string "(<)>" no está balanceado.

En la expresión además de los delimitadores pueden aparecer otros caracteres cuya presencia no influye en la condición de ser balanceado.

Se debe notar que el conjunto de pares de delimitadores se puede modelar con un mapping, en el que los delimitadores de entrada son las claves de las cuales los correspondientes delimitadores de cierre son sus valores asociados. El siguiente es un ejemplo de como construir el mapping:

```
mapping_t construr_map() {
  mapping_t parejas = crear_map();
  parejas = asociar('<', '>', parejas);
  parejas = asociar('(', ')', parejas);
  parejas = asociar('{', '}', parejas);
  parejas = asociar('[', ']', parejas);
  return parejas;
}
```

Sean `k` y `v` variables de tipo `char` y se consideran las siguientes condiciones:

```
es_clave(k, parejas)
es_valor(v, parejas)
valor(k, parejas) == v
```

La primera se cumple si y solo si `k` es delimitador de apertura. La segunda se cumple si y solo si `v` es delimitador de cierre. La tercera, asumiendo que `k` es delimitador de apertura, se cumple si y solo si `v` es el delimitador de cierre que corresponde a `k`.

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

- **uso_tads.cpp**

6.1. Plazos de entrega

El plazo para la entrega es el **13 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 *uso_tads.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 y mediante inspección de código. En todos los casos el programa deberá funcionar correctamente de acuerdo a la especificación proporcionada.

El resultado de la evaluación es la aprobación con hasta 2 puntos o la no aprobación, lo que implica la pérdida del curso.
La tarea otorgará 2 puntos solo si la entrega se aprueba en la primera instancia de evaluación. Si se aprueba en la segunda instancia (ver sección 6.6) puede otorgar 1 punto o ninguno. Los puntos obtenidos sirven para la aprobación o exoneración del curso. La adjudicación de los puntos 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 no satisfactorias

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**. La reentrega 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. 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 `principal.cpp`, `pila.cpp`, `mapping.cpp` y `uso_tads.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/pila.cpp -o obj/pila.o
$ g++ -Wall -Werror -Iinclude -g -c src/mapping.cpp -o obj/mapping.o
$ g++ -Wall -Werror -Iinclude -g -c src/uso_tads.cpp -o obj/uso_tads.o
$ g++ -Wall -Werror -Iinclude -g obj/principal.o obj/pila.o
  obj/mapping.o obj/uso_tads.o -o principal

$ ls
include  Makefile  obj  principal  principal.cpp  src  test

$ ls obj/
mapping.o  pila.o  principal.o  uso_tads.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 en la funcion `buscar` de `mapping.cpp` no hay instrucción `return` ya que por error se habría comentado la línea 39.

```
$ make
g++ -Wall -Werror -Iinclude -g -c src/mapping.cpp -o obj/mapping.o
src/mapping.cpp: In function ‘int buscar(char, mapping_t)’:
src/mapping.cpp:40:1: error: no return statement in function returning
non-void
 [-Werror=return-type] }
                       ^
cc1plus: all warnings being treated as errors
make: *** [obj/mapping.o] Error 1
```

Al volver a ejecutar make después de corregir el error solo se compila la que es necesario:

```
$ make
g++ -Wall -Werror -Iinclude -g -c src/mapping.cpp -o obj/mapping.o
g++ -Wall -Werror -Iinclude -g obj/principal.o obj/pila.o obj/mapping.o
  obj/uso_tads.o -o principal
```

es decir, no se vuelven a compilar `pila.cpp` ni `uso_tads.cpp` porque ya estaban correctamente compilados.

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/pila.cpp -o obj/pila.o
$ g++ -Wall -Werror -Iinclude -g -c src/mapping.cpp -o obj/mapping.o
$ g++ -Wall -Werror -Iinclude -g -c src/uso_tads.cpp -o obj/uso_tads.o
$ g++ -Wall -Werror -Iinclude -g obj/principal.o obj/pila.o
  obj/mapping.o obj/uso_tads.o -o principal
./principal < test/00.in > test/00.sal
./principal < test/01.in > test/01.sal
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
```

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

```
$ make testing
./principal < test/00.in > test/00.sal
./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 ----
-- CASOS CON ERRORES --
03
```

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

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

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

```
$ stat --print="%n  %s \n" test/*.diff
test/00.diff 0
test/01.diff 0
test/02.diff 0
test/03.diff 388
```

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 (`pila.o`, `mapping.o`, `uso_tads.o`, `principal.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`.