

Tarea 1

Introducción y Modularización

Curso 2024

La tarea consiste en la implementación de un *grupo de personas*. Un grupo de personas comprende una descripción y una colección de personas. De las personas interesa conocer el nombre, apellido, cédula y fecha de nacimiento. En un *grupo* se permite agregar, buscar y eliminar personas.

Para el modelado se implementará el tipo *TFecha*, el tipo *TPersona* y el tipo *TGrupo* que contiene un conjunto de personas.

Esta tarea tiene los siguientes objetivos específicos:

- Familiarizarse con el ambiente de trabajo del laboratorio.
- Realizar un primer acercamiento al lenguaje C*.
- Familiarizarse con la división del código de programas compuestos por módulos distribuidos en múltiples archivos.

La fecha límite de entrega es el **jueves 22 de agosto a las 16.00 horas**. El mecanismo específico de entrega se explica en la Sección 6. Por otro lado, para plantear **dudas específicas de cada paso** de la tarea, se provee un link a un **foro de dudas** al final de cada parte.

A continuación se presenta una **guía**, la cual se debe **seguir paso a paso** para resolver la tarea.

1. Armado de ambiente de programación

En esta sección se provee una guía de los pasos a seguir para contar con un ambiente de trabajo adecuado para la realización de la tarea. Está basada en el material *Armado del ambiente de programación para los laboratorios* de la sección [Recursos del Laboratorio](#) del sitio EVA del curso.

Se recomienda fuertemente seguir los pasos en detalle y **no avanzar al siguiente paso hasta completar correctamente el anterior**.

1. **Tener acceso a una distribución del sistema operativo Linux.** El sistema operativo que se utilizará durante el curso es Linux, y será el utilizado para corregir las tareas. Para acceder a un sistema Linux se puede seguir cualquiera de las siguientes opciones:
 - Trabajar en las máquinas de las salas estudiantiles, ya sea de forma física o remota. Para la conexión remota, debe instalar un cliente de Secure SHell (o SSH por sus siglas) y conectarse a *login-ens.fing.edu.uy* con su usuario informático de Facultad (*nombre.apellido*). En la mayoría de los sistemas operativos modernos ya se cuenta con un cliente SSH o puede instalar uno.
 - Utilizar una máquina virtual con Linux sobre el sistema de [Windows Services for Linux \(WSL\)](#) o sobre un software de virtualización.

Por lo tanto, antes de continuar deberán elegir alguna de estas opciones y seguir las indicaciones que se describen con mayor detalle en el EVA del curso de Programación 2, en la sección [Recursos del Laboratorio](#).

Si eventualmente cuenta con acceso a una máquina con otro sistema operativo (Windows, MacOS, etc) y desea trabajar en ese sistema, tenga en cuenta los siguientes puntos:

- No se responderán dudas de problemas específicos a otras distribuciones de sistema operativo que no sean Linux.
- En última instancia se corregirán las tareas en máquinas de la facultad con distribución Linux. Cualquier problema durante la corrección causado por trabajar en otro sistema operativo será interpretado como un error.

En este sentido, siempre que trabaje en otro ambiente, deberá verificar que su programa funciona correctamente en las máquinas de facultad antes de entregarlo.

Una vez que confirme que tiene acceso a Linux puede continuar con el siguiente paso. [Foro de dudas](#).

2. **Descargar materiales del EVA y descomprimirlos en la carpeta de desarrollo.** Los materiales para realizar cada tarea se extraen de un archivo específico para cada tarea que se encuentra en la sección [Laboratorio, Materiales y entrega](#) del sitio del curso.

Para desempaquetar el material se puede usar la utilidad `tar` desde la línea de comandos:

```
$ tar zxvf NombreArchivo.tar.gz
```

Los archivos específicos para cada tarea están dispuestos en una estructura de directorios *que debe conservarse*. Más información en [Funcionamiento del Laboratorio](#). [Foro de dudas](#).

3. **Tener acceso a un editor de texto o IDE adecuado.** El código se edita en el editor de texto o IDE preferido de cada estudiante. En las máquinas de Facultad se cuenta con el IDE Visual Studio Code y Eclipse. Para las tareas de programación de este curso se recomienda la utilización de Visual Studio Code o Sublime. No se responderán preguntas específicas sobre el uso de IDEs. [Foro de dudas](#).
4. **Compilar el código descargado.** El compilador es `g++`, que debe instalarse en caso de no estar instalado (ver [Funcionamiento del Laboratorio](#)). Recomendamos que la compilación y ejecución de los programas sea realizada desde una terminal del sistema. Para compilar, primero debe **posicionarse el directorio de trabajo de la terminal en la carpeta** `tarea1`, y **ejecutar** un comando de compilación:

```
$ cd <raiz_del_directorio_tarea1>/tarea1
$ g++ principal.cpp src/grupo.cpp src/persona.cpp src/fecha.cpp src/utills.cpp -o principal
```

Esto compila cada módulo, el `principal` y genera un ejecutable con nombre `principal`. [Foro de dudas](#).

5. **Compilar el código descargado utilizando `make`.** Para automatizar el proceso de desarrollo y prueba, se cuenta con el archivo `Makefile` (que está en la carpeta raíz de los materiales de descarga), que consiste en un conjunto de *reglas* para la utilidad `make`. Para ejecutar una regla específica se escribe en la terminal:

```
$ make nombre_regla
```

Las reglas del `make` permiten ejecutar una serie de comandos en la terminal de forma automática, sin tener que escribirlos uno por uno. Esto permite *compilar* de forma automatizada y luego *correr los casos de prueba* también de forma automatizada. Sin embargo, es siempre recomendable entender qué es lo que se está haciendo al utilizar la herramienta (ver [Instructivo Makefile](#)).

Por ejemplo, la regla `principal` compila `principal.cpp` y todos los archivos `.cpp` que se tengan que implementar como parte de la propuesta y luego genera el ejecutable `principal`. Esta regla es la predeterminada, es decir que se invoca si no se especifica ninguna regla. Por lo tanto, al **ejecutar**:

```
$ make
```

`Make` compilará cada módulo por separado y luego los enlazará en un ejecutable con nombre `principal`. Los archivos que resultan de la compilación de cada módulo, `(.o)` se mantienen en el directorio `obj`. Tener en cuenta que el comando de compilación dentro del `Makefile` contiene *banderas* que hacen que el compilador sea más estricto con la sintaxis del lenguaje. En este sentido, es posible que aparezcan errores de compilación que antes no sucedían y deben ser arreglados. [Foro de dudas](#).

6. **Probar el módulo principal.** El módulo principal incluye la función `main` que es la que se ejecuta al iniciar el programa. En esta función se encuentra implementado un *intérprete de comandos* que será utilizado para evaluar las distintas funciones de la tarea. Para ejecutar el principal debe ingresar en la terminal el comando:

```
$ ./principal
```

Luego ingrese el comando 'Fin', y como salida deberá ver el texto 'Fin.' y se cerrará el programa:

```
$ ./principal
1>Fin
Fin.
```

Foro de dudas.

7. **Modificar el módulo principal.** Como última prueba de la correcta configuración del ambiente, se realizará una pequeña modificación en el código, se compilará y ejecutará el programa para probar la modificación realizada. Cargue el código de toda la tarea en el editor elegido (por ejemplo, con VSCode puede hacer click derecho en el directorio `tarea1`, *Open with Code*) y abra el archivo `principal.cpp`. Tenga en cuenta que los editores recomendados permiten abrir el código de toda una carpeta, facilitando el acceso a los archivos.

Una vez abierto el archivo `principal.cpp`, revise el código y encuentre donde se realiza la acción asociada al comando 'Fin'. **Modifique el código** para que en lugar de imprimir 'Fin.' se imprima el texto 'Chau mundo.'. Compile, ejecute e ingrese el comando 'Fin'. Asegúrese que la salida es el nuevo texto ingresado. Por último, **vuelva el código a la versión original**. [Foro de dudas.](#)

2. Entendiendo los módulos de la tarea

Cuando la dimensión o complejidad del programa a construir crece, conviene dividir el código en varios archivos, incluyendo en cada uno entidades (tipos, constantes, funciones) altamente relacionadas entre sí. A estos agrupamientos los denominaremos *módulos*. Puede encontrar mas detalles sobre este tema en [Modularización](#).

La tarea consiste de 4 módulos: `utils`, `fecha`, `persona` y `grupo`, además de un archivo `principal`. 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. Esto 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` (la letra proviene de *headers*, encabezados) y a los segundos con la extensión `.c` o `.cpp`.

En la tarea, los archivos de los módulos están ordenados de la siguiente forma:

- En la raíz se encuentra el módulo principal (**principal.cpp**), el **Makefile** y el ejecutable que se generará tras la compilación.
 - Los archivos de encabezado (**.h**, *headers*) están en el directorio **include**.
 - Los archivos a implementar (**.cpp**) están en el directorio **src**.
 - Los archivos que resultan de la compilación de cada módulo, (**.o**), se mantienen en el directorio **obj**.
1. **Abra y examine los archivos `fecha.h` y `fecha.cpp` descargados.** Observe que en `fecha.h` se declara el tipo `TFecha` como un puntero al struct `rep_fecha` que aún no está definido:

```
typedef struct rep_fecha *TFecha;
```

Además, se declaran varias funciones. Lo que se declara en `fecha.h` debe implementarse en `fecha.cpp`, por lo que en `fecha.cpp` se debe incluir el cabezal:

```
#include "../include/fecha.h"
```

Observe que en `fecha.cpp` se encuentra la definición del struct `rep_fecha`. **Complete el struct** de la siguiente manera:

```
struct rep_fecha {
    nat dia, mes, anio;
};
```

(el tipo `nat` está definido `utils.h`). [Foro de dudas](#).

3. Implementación del módulo Fecha

El módulo Fecha es el que contiene la representación de fecha a utilizar y un conjunto de operaciones para interactuar con esta representación. A continuación se presentan los pasos a seguir para su implementación y prueba:

1. **Implementar la función `crearTFecha`**. Ubicados en el archivo `fecha.cpp`, se agregará el código necesario para crear una nueva instancia de una fecha de tipo `TFecha`. Observe que entre las operaciones del módulo `fecha`, algunas devuelven elementos de tipo `TFecha`. Como estos son punteros, la memoria a la que apuntan se obtiene de manera dinámica. Esto se realiza con el operador `new`.

Por lo tanto, la primera línea de código a agregar será:

```
nuevaFecha = new rep_fecha;
```

Luego, se debe asignar los valores pasados por parámetro a la función a cada campo de la nueva fecha. Para eso, se utilizará el operador `->` para acceder al campo del struct al que apunta el puntero `nuevaFecha`:

```
nuevaFecha->dia = dia;
```

(haga lo mismo para los campos `mes` y `anio`). Por último, queda retornar la nueva fecha creada utilizando la directiva `return`. [Foro de dudas](#).

2. **Verificar errores de compilación y ejecución**. Una vez realizados los cambios anteriores, **compile la tarea** como se explicó en el punto 1. Luego, **ejecute el principal** y pruebe que el comando `crearFecha 25/2/2024` ejecuta sin errores.

```
$ ./principal
$ 1>crearFecha 20/2/2024
```

Notar que en este momento todavía no es posible verificar que la fecha se creó correctamente, para esto se debe avanzar un poco más y definir un caso de prueba. Finalice la ejecución del programa con el comando `Fin`. [Foro de dudas](#).

3. **Implementar `liberarTFecha`**. En el lenguaje C* la memoria que se obtiene dinámicamente mediante `new` debe ser liberada con la instrucción `delete`. Esto debe hacerse de manera sistemática: por cada instrucción que solicita memoria debe haber alguna o algunas correspondientes que la liberen, tal vez en otra función. En este caso, esto se realiza en la función `liberarTFecha` de la siguiente manera:

```
delete fecha;
fecha = NULL;
```

En general, es recomendable asignar el puntero a NULL luego de liberada la memoria de forma de que apunte a un valor conocido. Al igual que en el paso anterior, realice una verificación sencilla compilando y probando con un caso de prueba sencillo:

```
1>crearFecha 1/2/2024
2>liberarFecha
3>Fin
```

Avance al siguiente punto si el caso anterior ejecutó sin errores. [Foro de dudas](#).

4. **Detección de errores de memoria.** Dada la complejidad que implica la gestión de memoria que debe realizar el programador, es siempre recomendable controlar que el programa hace un correcto manejo de esta. Una vez que verificado que el programa cumple la especificación funcional (lo que el programa debe hacer), se debe correr alguna herramienta de análisis del uso de memoria para verificar que no hay problemas de manejo de memoria. Una herramienta que permite realizar esta verificación es [valgrind](#). Las indicaciones de cómo instalar valgrind se encuentran en: [Funcionamiento del Laboratorio](#) y algunos ejemplos de uso en [Detección de errores en uso de memoria](#).

A continuación se generará una pérdida de memoria (memory leak) al crear una fecha y finalizar el programa sin liberarla. Se utilizará valgrind para que muestre el problema.

Ejecute el principal con valgrind:

```
$ valgrind ./principal
```

Luego ejecute los comandos para crear una fecha y el comando Fin:

```
1>crearFecha 21/2/2024
2>Fin
```

Debería verse una salida como la siguiente, donde se muestra que se deja sin liberar un bloque de 12 bytes:

```
$ valgrind ./principal
==330925== Memcheck, a memory error detector
==330925== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==330925== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==330925== Command: ./principal
==330925==
1>crearFecha 21/2/2024
2>Fin
Fin.
==330925==
==330925== HEAP SUMMARY:
==330925==      in use at exit: 12 bytes in 1 blocks
==330925==    total heap usage: 5 allocs, 4 frees, 74,776 bytes allocated
==330925==
==330925== LEAK SUMMARY:
==330925==      definitely lost: 12 bytes in 1 blocks
==330925==      indirectly lost: 0 bytes in 0 blocks
==330925==      possibly lost: 0 bytes in 0 blocks
==330925==      still reachable: 0 bytes in 0 blocks
==330925==      suppressed: 0 bytes in 0 blocks
==330925== Rerun with --leak-check=full to see details of leaked memory
==330925==
==330925== For lists of detected and suppressed errors, rerun with: -s
==330925== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Realice la misma prueba, pero ahora agregando el comando `liberarFecha` antes de terminar el programa. Debería ver una salida sin pérdidas de memoria:

```
==378400== Memcheck, a memory error detector
==378400== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==378400== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==378400== Command: ./principal
==378400==
1>crearFecha 21/2/2024
2>liberarFecha
3>Fin
Fin.
==378400==
==378400== HEAP SUMMARY:
==378400==   in use at exit: 0 bytes in 0 blocks
==378400==   total heap usage: 5 allocs, 5 frees, 74,776 bytes allocated
==378400==
==378400== All heap blocks were freed -- no leaks are possible
==378400==
==378400== For lists of detected and suppressed errors, rerun with: -s
==378400== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Foro de dudas.

5. **Implementar** `imprimirTFecha`. La fecha se debe imprimir con el formato `dia/mes/año`, por ejemplo, `2/3/2024`. Para esto se utiliza la función `printf`¹ con las especificaciones de formato correspondientes. Tener en cuenta que en la impresión de la fecha no deben haber espacios (ni antes, ni después) y después de la fecha se debe imprimir el caracter de fin de línea `'\n'`. Pruebe su implementación **ejecutando el programa principal** e ingresando una secuencia de comandos para crear una fecha, imprimirla (con el comando `imprimirFecha`) y luego liberarla. Recuerde que luego de cada cambio siempre debe volver a compilar. **Foro de dudas.**
6. **Uso de casos de prueba.** Los casos de prueba están en el directorio `test`. Los archivos con extensión `.in` (entradas) contienen una secuencia de comandos. Por cada uno de estos archivos hay uno con extensión `.out` (salidas) que contiene la respuesta esperada del programa. A continuación se explica cómo ejecutar un caso de prueba de forma manual:
 - Abra el archivo `fecha1-crear-imprimir-liberar.in`.
 - Ejecute el programa principal e ingrese los comandos que se listan en el archivo anterior. Es decir, copie una a una las líneas e ingréselas al principal.
 - Abra el archivo `fecha1-crear-imprimir-liberar.out` y compare la salida. Si la salida coincide, esto significa que el test realizado es correcto.

Ahora se verá cómo ejecutar un caso de prueba sin necesidad de ingresar cada comando de forma manual. Para esto se redirigirá la entrada y salida del programa con los caracteres `'<'` y `'>'` en la línea de comandos. **Ejecute** el programa principal utilizando como entrada el archivo `.in`:

```
$ ./principal < test/fecha1-crear-imprimir-liberar.in
```

Esto provoca que las entradas leídas por el programa se tomen desde el archivo en lugar de desde la consola de comandos (entrada estándar). Es equivalente a ingresar el texto del `.in` línea por línea, y genera una salida (si no hubo errores) que se puede verificar que coincida con el archivo `.out`. Por último, en vez de comparar a ojo las salidas, se puede realizar la comparación de forma automática y más rigurosa, guardando la salida del programa en un archivo y luego utilizando el comando `diff` para compararla con el `.out`. **Ejecute** los siguientes comandos:

¹<https://cplusplus.com/reference/cstdio/printf/>

```
$ ./principal < test/fecha1-crear-imprimir-liberar.in > test/↵
  salidas/fecha1-crear-imprimir-liberar.sal
$ diff test/fecha1-crear-imprimir-liberar.out test/salidas/↵
  fecha1-crear-imprimir-liberar.sal
```

Si los archivos son iguales, no se imprime nada. Si son diferentes, se muestran las líneas en las que difieren y el contenido de ellas, y se determina que **hay un error** en el programa. [Foro de dudas](#).

7. **Casos de prueba automatizados.** Para correr los mismos comandos del paso anterior de forma automatizada **ejecute**:

```
$ make test/salidas/fecha1-crear-imprimir-liberar.diff
```

Los archivos generados al hacer las pruebas de esta forma van al directorio **test/salidas**. Los archivos con extensión `.sal` contienen el resultado de la ejecución. Los archivos con extensión `.diff` contienen la comparación del `.sal` con el `.out` correspondiente. De manera abreviada se puede ejecutar **\$ make t-NN**, donde NN es el nombre del caso de prueba. Por ejemplo, **pruebe ejecutar**:

```
$ make t-fecha1-crear-imprimir-liberar
```

Este comando realizará la ejecución del caso de prueba y mostrará en pantalla el resultado. Si la salida coincide con la esperada, mostrará la palabra **'Bien'**; si no, mostrará las líneas donde encontraron diferencias y dirá que hay un **ERROR**. En este caso no se generan los archivos de salida y diff.

En caso de encontrar un error o una diferencia al ejecutar un caso de prueba de esta forma, se recomienda revisar en detalle el contenido del caso de prueba y ejecutarlo de forma manual como se mostró anteriormente para encontrar el error.

Por último, luego de los anteriores comandos se puede ejecutar, además, el siguiente comando, para verificar el correcto uso de la memoria y ser notificado de posibles errores de memoria:

```
$ valgrind --leak-check=full ./principal < test/NN.in > test/salidas/NN.sal
```

[Foro de dudas](#).

8. **Generar una violación de segmento.** Una *violación de segmento* (o *segmentation fault* en inglés) es un error común que se produce cuando el programa intenta acceder a un lugar de la memoria que no está permitido. Esto suele suceder, por ejemplo, cuando se quiere acceder a una variable que es NULL, o que aún no fue inicializada. Para reproducir este error, primero **pruebe ejecutar el programa principal e ingresar directamente el comando *imprimirFecha***:

```
$/principal
$1> imprimirFecha
```

El programa debería imprimir la siguiente salida:

```
principal: principal.cpp:194: void main_imprimirFecha(TFecha):
Assertion 'fecha != NULL' failed.
```

Lo que dice ese mensaje es que, en la línea 194 de *principal.cpp*, el programa evaluó que la fecha fuera distinta de NULL y el resultado fue falso, lo que ocasionó que el programa abortara su ejecución. La línea 194 del *principal.cpp* es la siguiente:

```
194| assert(fecha != NULL);
```


donde la función `assert` permite controlar que se cumpla una condición, y, en caso de no cumplirse, abortar la ejecución del programa. En este caso, como se quiere imprimir una fecha, hay que asegurar que la fecha no sea `NULL` para acceder a sus valores. Para probar qué sucedería si no se realiza este chequeo mediante `assert`, **pruebe comentar la línea 194, compilar el programa y ejecutar**:

```
$. /principal
$1> imprimirFecha
```

Ahora debería producirse una salida similar a la siguiente:

```
Violación de segmento ('core' generado)
```

En este caso ocurrió una violación de segmento durante la ejecución del programa al querer acceder a los campos de una fecha que es `NULL`. Sin embargo, el error en tiempo de ejecución no señala por qué se produjo el error, ni en qué línea ocurrió, y esto puede llevar a que sea difícil de encontrar cómo solucionarlo. Para facilitar el proceso de corrección **ejecute el programa con `valgrind`**.

```
$valgrind ./principal
```

En este caso, parte de la salida del programa debería ser similar a la siguiente:

```
==187097== Memcheck, a memory error detector
==187097== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==187097== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==187097== Command: ./principal
==187097==
1>imprimirFecha
==187097== Invalid read of size 4
==187097==    at 0x4020E7: imprimirTFecha(rep_fecha*) (fecha.cpp:87)
==187097==    by 0x40170C: main_imprimirFecha(rep_fecha*) (principal.cpp:194)
==187097==    by 0x4012DC: main (principal.cpp:97)
==187097== Address 0x8 is not stack'd, malloc'd or (recently) free'd
==187097==
==187097==
==187097== Process terminating with default action of signal 11 (SIGSEGV): dumping core
...
```

En esta salida se puede ver inicialmente que el error que se produjo fue un *invalid read* (marcado en marrón), o lectura inválida, y se explica en la línea verde que en esa dirección (*Address 0x8*) la variable no tenía asignada memoria válida. En las líneas azul, naranja y roja, se muestra una secuencia a la que nos referimos como *stacktrace* (o *backtrace*), que es la secuencia de funciones que se encontraba activa cuando se produjo el error. Es decir, en este caso, el error se produjo mientras se ejecutaba la función `imprimirTFecha` (marcada en rojo), específicamente cuando se ejecutaba la línea 87 del archivo `fecha.cpp`, y esta función fue llamada desde `main_imprimirFecha`, en la línea 195 de `principal.cpp`, y esta función fue llamada desde la función `main`, en la línea 97 de `principal.cpp`. En definitiva, con esta información se puede ir a la línea 87 de `fecha.cpp`, y ver exactamente qué línea del programa produjo el error de lectura inválida. Tener en cuenta que la línea 87 es específica del código de prueba del grupo docente de laboratorio, y la suya puede ser una línea distinta.

Por último, **vuelva a descomentar la línea 194 de `principal.cpp`**. [Foro de dudas](#).

9. **Implementar la función `incrementarTFecha`**. En este punto el objetivo es implementar una función que, dada una fecha y una cantidad (mayor o igual a cero) de días, aumente la fecha en esa cantidad de días. Esto implica no solo aumentar el campo `dia` de una fecha, sino que en algunos casos será necesario aumentar el campo `mes` e incluso puede ser necesario aumentar el campo `anio`. A continuación se presenta parte del código de esta función:


```

// Función para aumentar una fecha en una cantidad dada de días
void aumentarTFecha(TFecha &fecha, nat dias) {
    fecha->dia += dias;
    while (fecha->dia > diasMes(fecha->mes, fecha->anio)) {
        fecha->dia -= diasMes(fecha->mes, fecha->anio);
        fecha->mes++;
        if (fecha->mes > 12) {
            fecha->mes = 1;
            fecha->anio++;
        }
    }
}
}

```

Como se puede observar, esta función utiliza una función auxiliar denominada `diaMes`. El uso de funciones auxiliares es algo muy común y recomendable cuando partes del problema a resolver pueden descomponerse en problemas más chicos. En C no se puede definir una función anidada dentro de otra, como se puede hacer en Pascal. Por lo tanto, la función auxiliar debe definirse fuera de las funciones en las que se va a usar. Además, la declaración debe estar antes de que la función sea usada.

Como puede ver, en este caso, la función auxiliar `diaMes` se utiliza para obtener la cantidad de días de el mes de una fecha. Por lo tanto, deberá implementar esta función:

```

// Función para obtener la cantidad de días de un mes en un año dado
nat diasMes(nat mes, nat anio)

```

Tenga en cuenta que, para el mes de febrero, la cantidad de días depende de si el año es o no **bisiesto**. Puede ser necesaria una segunda función auxiliar para determinar si un año es bisiesto.

Luego de resuelto el punto anterior, es momento de **probar la implementación realizada**. Para esto corra el caso de prueba `fecha2-aumentar`. [Foro de dudas](#).

10. **Implementar la función** `compararTFechas`. A partir de la especificación de la función dada en `fecha.h`, implemente el código de la función y luego ejecute el caso de prueba `fecha3-comparar`. [Foro de dudas](#).
11. **Ejecutar el caso de prueba** `fecha4-combinado`. Este es un caso de prueba que combina la utilización de todas las funciones de fecha. Si encuentra algún error o alguna discrepancia en la salida, deberá seguir los pasos descritos en el paso 7 para encontrar el problema. [Foro de dudas](#).

4. Módulo Persona

En esta sección se realizará la implementación del módulo Persona. A diferencia de la sección anterior, en este caso la guía no será tan detallada, asumiendo que todas las indicaciones dadas anteriormente serán utilizadas. En particular, es importante seguir el proceso de implementación, compilación y prueba cada vez que se considere necesario. **Recuerde leer la definición de cada función en el archivo .h antes de implementarla.**

1. **Definir la estructura para una persona.** En este paso, se debe implementar el `struct rep_persona` que contendrá la información de una persona. Esta estructura debe contener una cédula de tipo entero, un nombre de tipo texto, un apellido de tipo texto y una fecha de nacimiento de tipo `TFecha`. Tanto para el nombre como para el apellido se debe utilizar un arreglo de caracteres (`char`) de tamaño `MAX_NOMBRE` y `MAX_APELLIDO` respectivamente. [Foro de dudas](#).
2. **Implementar la función** `crearTPersona`. Al igual que para el caso de Fecha, es necesario solicitar memoria para un nuevo elemento de tipo `TPersona`. En este punto, se debe tener especial cuidado en el manejo de los campos de texto (`nombre` y `apellido`), ya que son de tipo arreglo. Para asignar

el nombre y apellido recibidos como parámetro al campo de la persona, se debe utilizar la función `strcpy`²:

```
strcpy(nuevaPersona-><parametro_nombre>, nombre);
```

Los detalles de por qué esto es necesario se verán más adelante en el curso. [Foro de dudas.](#)

3. **Implementar las funciones** `nombrePersona`, `apellidoPersonay` `liberarTPersona`, y **ejecutar el caso de prueba** `persona1-crear-imprimirnombre-liberar` [Foro de dudas.](#)
4. **Implementar las funciones** `cedulaTPersona` y `fechaNacimientoTPersona`, y **ejecutar los casos de prueba** `persona2-imprimirci` y `persona3-imprimirfecha`. [Foro de dudas.](#)
5. **Implementar la función** `imprimirTPersona`. Esta función debe imprimir en pantalla la información de la persona de la siguiente forma:

```
Persona <nombre> <apellido>
Cedula: <cedula>
Fecha de nacimiento: <fecha de nacimiento>
```

Ejecuta el caso de prueba `persona4-imprimirpersona`. [Foro de dudas.](#)

6. **Implementar la función** `esMasJoven` y **ejecutar los casos de prueba** `persona5-esmasjovenpersona` y `persona6-esmasviejapersona`. [Foro de dudas.](#)

5. Módulo Grupo

En esa sección realizaremos la implementación del módulo Grupo. **Recuerde leer la definición de cada función en el archivo .h antes de implementarla.**

1. **Implementar** `struct rep_grupo` **como un arreglo con tope**, donde el tamaño del arreglo está dado por `MAX_PERSONAS`. [Foro de dudas.](#)
2. **Implementar las funciones** `crearTGrupo`, `agregarAGrupo`, `imprimirTGrupo` y `liberarTGrupo`. Recuerde que la grupo mantiene las personas de forma ordenada de **menor a mayor** por fecha de nacimiento. Si dos personas tienen la misma fecha de nacimiento, debe aparecer primero en el grupo la persona ingresada de forma más reciente. Por otro lado, el formato en el que se debe imprimir el Grupo es simplemente utilizando de forma secuencial la función `ImprimirTPersona`. **Ejecutar los casos de prueba** `grupo1-crear-liberar`, `grupo2-agregar-imprimir` y `grupo3-agregarvarios-imprimir`. [Foro de dudas.](#)
3. **Implementar la función** `estaEnGrupo` y **ejecutar el caso de prueba** `grupo4-buscar`. [Foro de dudas.](#)
4. **Implementar la función** `removerDeGrupo` y **ejecutar el caso de prueba** `grupo5-remover`. [Foro de dudas.](#)
5. **Implementar las funciones** `imprimirPersonasFecha` y `hayPersonasFecha`. **Ejecutar los casos de prueba** `grupo6-personasfecha-hay-imprimir`. [Foro de dudas.](#)
6. **Ejecutar el caso de prueba** `grupo7-combinado`. [Foro de dudas.](#)

²<https://cplusplus.com/reference/cstring/strcpy/>

6. Test final y entrega de la tarea

Para finalizar con la prueba del programa utilice la regla *testing* del Makefile y verifique que no hay errores en los tests públicos. La regla *testing* corre todos los tests uno a uno verificando que no haya errores. Esta regla se debe utilizar **únicamente luego de realizados todos los pasos anteriores**.

1. Ejecutar el comando:

```
$ make testing
```

La regla *testing* tiene en cuenta el uso de memoria y permite probar todos los casos de prueba. En el directorio **test/salidas** quedan los archivos **.sal** y **.diff** correspondientes a cada uno de los casos. **Se sugiere usar esta regla solo en este momento, después de haber probado cada caso individualmente, para confirmación.** Si la salida no tiene errores, al final se imprime lo siguiente:

```
-- RESULTADO DE CADA CASO --  
111111111111111111
```

Donde un 1 simboliza que no hay error y un 0 simboliza un error en un caso de prueba, en este orden:

```
fecha1-crear-imprimir-liberar  
fecha2-aumentar  
fecha3-comparar  
fecha4-combinado  
persona1-crear-imprimirnombre-liberar  
persona2-imprimirci  
persona3-imprimirfecha  
persona4-imprimirpersona  
persona5-esmasjovenpersona  
persona6-esmasviejapersona  
grupo1-crear-liberar  
grupo2-agregar-imprimir  
grupo3-agregarvarios-imprimir  
grupo4-buscar  
grupo5-remove  
grupo6-personasfecha-hay-imprimir  
grupo7-combinado
```

[Foro de dudas.](#)

2. **Prueba de nuevos tests.** Si se siguieron todos los pasos anteriores el programa creado debería ser capaz de ejecutar todos los casos presentados en los tests públicos. Para asegurar que el programa es capaz de ejecutar correctamente ante nuevos casos de uso es importante realizar tests propios, además de los públicos. Para esto **Cree un nuevo archivo en la carpeta test**, con el nombre *test_propio.in*, y **escriba una serie de comandos** que permitan probar casos de uso que no fueron contemplados en los casos públicos. **Ejecute el test** mediante el comando:

```
$ ./principal < test/test_propio.in
```

y verifique que la salida en la terminal es consistente con los comandos ingresados. La creación y utilización de casos de prueba propios es fundamental para robustecer el programa para la prueba de los casos de test privados. [Foro de dudas.](#)

3. **Prueba en pcunix.** Es importante probar su resolución de la tarea con los materiales más recientes y en una pcunix, que es el ambiente en el que se realizarán las correcciones. Para esto siga el procedimiento explicado en [Sugerencias al entregar.](#) [Foro de dudas.](#)

4. **Armado del entregable.** El archivo entregable final debe generarse mediante el comando:

```
$ make entrega
```

Con esto se empaquetan los módulos implementados y se los comprime generando el archivo `EntregaTarea1.tar.gz`.

El archivo a entregar **DEBE** ser generado mediante este procedimiento. Si se lo genera mediante alguna otra herramienta (por ejemplo, usando un entorno gráfico) **la tarea no será corregida**, independientemente de la calidad del contenido. Tampoco será corregida si el nombre del archivo se modifica en el proceso de entrega. [Foro de dudas](#).

5. **Subir la entrega al receptor.** Se debe entregar el archivo **EntregaTarea1.tar.gz**, que contiene los módulos a implementar **fecha.cpp**, **persona.cpp** y **grupo.cpp**. Una vez generado el entregable según el paso anterior, es necesario subirlo al receptor ubicado en la sección Laboratorio del EVA del curso. **Recordar que no se debe modificar el nombre del archivo generado mediante** `make entrega`. Para verificar que el archivo entregado es el correcto se debe acceder al receptor de entregas y hacer click sobre lo que se entregó para que automáticamente se descargue la entrega. [Foro de dudas](#).