SSL 2005.1 – TP Simulación de un Año Académico de SSL – Entrega #4 – TAD CalendarioAcademico y Programa de Simulación

por José María Sola y Jorge Muchnik.

1.0.0.20050616

Requisitos

- Haber aprobado TP #0.
- Haber realizado las Entregas #1, #2 y #3.
- Tener correctamente implementados y probados los TADs **Actividad**, **Curso** y **MesaDeExamenFinal**.
- Haber leído y comprendido el documento anterior: "**SSL 2005.1 – Presentación General del Trabajo Práctico – Simulación de un Año Académico de SSL**"

Objetivos

- **Especificar** los *valores* y *operaciones* del TAD **CalendarioAcademico** aplicando las herramientas conocidas.
- **Implementar** el TAD **CalendarioAcademico** en una **Biblioteca** aplicando las herramientas conocidas.
- **Probar** la implementación del TAD **Curso** mediante una **aplicación de prueba**.
- Implementar el TAD **CalendarioAcadémico** en ANSI C utilizando **semántica de referencia** en contraste con la **semántica de valor** aplicada en los tres anteriores TADs.
- Construir el **programa de simulación** que integra los cuatro TADs diseñados.
- Aplicar **archivos binarios** y sus **streams** (**flujos de datos**) asociados de ANSI C.
- Aplicar la función `strtok` para *tokenizar* cadenas.
- Aplicar **Autómatas Finitos** como **reconocedor** de cadenas.
- **Capturar** las **salidas** de los **procesos de traducción** y de la **aplicación de prueba** y del **programa de simulación**.
- Obtener una **comprensión sistémica del TP** y **revisar** los conceptos de **ADT** y su **implementación en ANSI C** obtenidos a lo largo del TP.

Parte #1 – TAD CalendarioAcademico

Sobre el TAD CalendarioAcademico

Importante: Esta sección es un relato que describe el TAD; **no es su especificación**.

El TAD **CalendarioAcademico** define el **cronograma** de **actividades** relacionadas con SSL que se realizarán a lo largo de un **año académico**. Cada año académico comienza el 1ero de Abril de un determinado año y finaliza el 31 de Marzo del año siguiente, el calendario se divide en dos cuatrimestres, de 16 semanas cada uno. Para un día determinado puede haber una o ninguna actividad asignada.

Un valor del TAD CalendarioAcademico es un **año académico**, una **secuencia finita de valores del TAD Actividad** y un indicador de la **actividad actual**. Por lo que un valor del CalendarioAcademico es la tres-upla: (**Año**, **Actividades**, **Actividad actual**). *Actividades* es un conjunto de 365 ó 366 actividades, dependiendo del *año*; este conjunto está ordenando ascendentemente por la fecha de la actividad, con la primer actividad ocurriendo el 1ero de Abril. *Actividad actual* es la actividad con la que se está trabajando actualmente.

Operaciones

Las operaciones relacionadas con un valor del TAD CalendarioAcademico son: Crear un CalendarioAcademico, asignar una actividad a un CalendarioAcademico, obtener el año de un CalendarioAcadémico, obtener la actividad actual de un CalendarioAcadémico y obtener la siguiente actividad.
La especificación de cada operación debe incluir todas las precondiciones y poscondiciones. Realizar un análisis correcto y exhaustivo de estas condiciones para una especificación correcta y sin ambigüedades de las operaciones.

Creación

- **Creación**. Dados un año (Natural) produce un valor del TAD CalendarioAcademico, con el conjunto de actividades vacío y la actividad actual indefinida. Utilizar un arreglo de 366 valores del TAD Actividad. La implementación debe alocar dinámicamente la estructura que implementa un valor de este TAD y retornar un puntero a esta estructura (*semántica de referencia*). La actividad actual está representada por un entero, índice del arreglo Actividades.

Acceso

- **Año**. Dado un valor del TAD CalendarioAcademico, retorna su año.
- **ActividadActual**. Dado un valor del TAD CalendarioAcademico, retorna la actividad actual. La implementación debe retornar un puntero a esa actividad.

Modificación

- **Asignar actividad**. Dados un valor del TAD CalendarioAcademico y un valor del TAD Actividad, retorna un nuevo valor del TAD CalendarioAcademico, pero con su conjunto de actividades modificado. Si la Actividad dato tuviese como *fecha 29 de febrero* y el año no fuese *bisiesto*, el valor del TAD CalendarioAcademico resultado es idéntico al dato. La implementación recibe un puntero a un objeto del tipo Actividad, y copia el contenido de ese objeto a la posición correspondiente dentro del arreglo de actividades.
- **Ir a siguiente actividad**. Avanza hasta la siguiente actividad que no sea del tipo *SinActividad*. Dados un valor del TAD CalendarioAcademico, retorna dos valores: 1) un nuevo valor del TAD CalendarioAcademico, pero con una actividad actual diferente, la cual no debe ser del tipo *SinActividad*, y 2) la distancia (N_0) entre la actividad actual del CalendarioAcademico dato y la actividad actual del CalendarioAcademico resultado.
  Si la distancia fuese 1 significa que hubo una actividad contigua a la otra. Si la distancia fuese mayor a 1, significa que entre actividad y actividad hubo días sin actividad, siendo la cantidad de días sin actividad determinada por la distancia menos 1. Si la distancia fuese cero, significa que es la última actividad del año académico.

Requeridas para esta la Implementación

- **Destrucción**. Dada una variable del TAD CalendarioAcademico, libera los recursos previamente asignados, incluyendo los tomados por cada Actividad del arreglo de actividades.

Restricciones y Guías para la Implementación del TAD CalendarioAcademico

```
typedef struct {
    int       anio;
    Actividad actividades[366];
    int       actividadActual;
} CalendarioAcademico;
```

```
1. CalendarioAcademico* CalendarioAcademico_Crear(
     int unAnio /* in */
   );

2. void CalendarioAcademico_Destruir(
     const CalendarioAcademico*
     unCalendarioAcademico /* in */
   );
```

```
3. int CalendarioAcademico_GetAnio(
     const CalendarioAcademico* unCalendarioAcademico /* in */
   );

4. Actividad* CalendarioAcademico_GetActividadActual(
     const CalendarioAcademico* unCalendarioAcademico /* in */
   );

5. void CalendarioAcademico_SetActividad(
     CalendarioAcademico* unCalendarioAcademico, /* inout */
     const Actividad*     unaActividad           /* in */
   );

6. int CalendarioAcademico_MoveNextActividad(
     CalendarioAcademico* unCalendarioAcademico /* inout */
   );
```

Parte #2 – Programa de Simulación

¿Qué es una Simulación por Computadora?

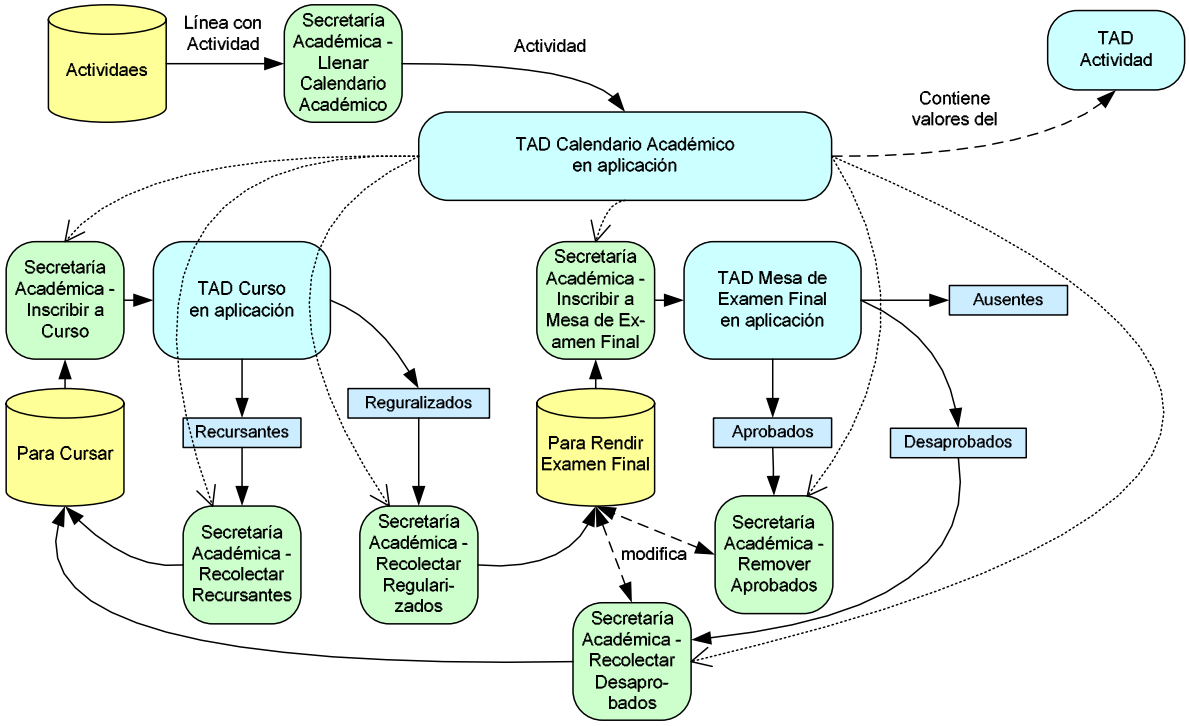
La simulación intenta representar ciertas características del comportamiento de un sistema mediante el comportamiento de otro sistema. Para ello se diseña un modelo, mediante abstracción e hipótesis de simplificación, que representa el sistema a simular. La simulación permite estudiar el comportamiento de un sistema todavía inexistente, o de un sistema existente que no se desea o que no se puede acceder o modificar. El estudio de un sistema mediante su simulación produce información útil para la toma de decisiones.
La simulación por computadora permite la recreación de una secuencia de eventos reales a través de la ejecución de un programa. La aleatoriedad de la vida real puede ser representada por técnicas que producen números pseudo-aleatorios.

Sobre el Programa de Simulación

El programa de simulación será un "Programa comando". Este programa comando será un programa no interactivo, sus entradas se determinadas al momento de ejecutar el programa (argumentos).
Esta simulación lee los eventos de un archivo con el calendario académico y atiende cada uno de los eventos y sub-eventos. Escribe en la salida standard un mensaje descriptivo por cada evento y sub-evento atendido, junto con posibles resultados.
El calendario académico, los alumnos en condiciones de cursar y los alumnos en condiciones de rendir examen final se encuentran en archivos cuyos nombres son los últimos tres argumentos del programa comando (ver más adelante la sintaxis del programa comando).
A parte de modificar el archivo de los alumnos en condiciones de cursar y el archivo de los alumnos en condiciones de rendir examen final, la ejecución de esta simulación produce los archivos generados por la ejecución de las operaciones de los TADs Curso y MesaDeExamenFinal.

Sobre el Diseño del Programa de Simulación

El diseño de la simulación se basa sobre un **modelo simplificado** de un año académico de SSL. Se simulará las actividades de un curso del primer cuatrimestre y uno del segundo cuatrimestre como así también de los 10 exámenes finales del año. El eje central y elemento definitorio es el **Calendario Académico**. En él, se establece, día a día, las **actividades** programadas para SSL. La componente estadística está dada por la aleatoriedad de las inscripciones, asistencia y aprobaciones.
El siguiente diagrama modela, en el **contexto de una aplicación** y utilizando los TADs previamente diseñados, el sistema de simulación.
Se representan las entidades que actúan como repositorio de datos: el conjunto de alumnos en **condición de cursar**, el conjunto de alumnos en **condición de rendir final** que incluye un **contador de veces que rindió final** y el conjunto de **actividades**. Las tres entidades serán implementadas como **archivos de texto**.
Se representan los **archivos que producen algunas de las operaciones** de los TADs Curso y MesaDeExamenFinal.
Se indican los procesos que la **Secretaría Académica** de esta facultad realiza para poner en movimiento la información que permite pasar un alumno de un estado a otro y llevar a cabo las diferentes actividades académicas. Estos procesos serán implementados como funciones del programa de simulación.
Se representa con líneas de puntos el control que el Calendario Académico tiene sobre el **disparo** (comienzo) de los procesos.



Sintaxis del Programa Comando

```
SimularSsl anio ca ecdc ecdref
```

anio: Representa un número entero en base 10 que indica el año a simular.
ca: Calendario Académico. Nombre de un archivo de texto con 365 ó 366 líneas (si el año es bisiesto) con una actividad por línea, con los siguientes cuatro campos separados por caracteres de tabulado: día, mes corto (tres letras: "Ene", ..., "Dic"), código de dos letras que representa el tipo de actividad, descripción. Los códigos de los tipos de actividad son: SinActividad: SA, ClaseNormal: CN, EvaluacionGrupalEscrita: GE, EvaluacionGrupalOral: GO, EvaluacionIndividual: EI, ExamenFinal: EF, InscripcionACurso: IC, InscripcionAExamenFinal: IF, PrimeraClase: PC, PrimerRecuperatorio: PR, SegundoRecuperatorio: SR, UltimaClase: UC.
ecdc: En condiciones de cursar. Nombre de un archivo de texto, con un legajo por línea.
ecdref: En condiciones de rendir examen final. Nombre de un archivo de texto. Cada línea contiene dos campos separados por un espacio: la cadena que representa el legajo y un entero que representa la cantidad de veces que rindió el examen final (0, 1, 2, ó 3).

Si el programa comando es invocado sin argumentos, se escribe un mensaje de ayuda en el flujo `stdout`. Si el programa se invoca con una cantidad de argumentos incorrecta, o los archivos no se pueden abrir, leer o escribir, u ocurre alguna excepción, se escribe un mensaje de error en el flujo `stdout`.

Ejemplo 1.
```
SimularSsl 2001 Calendario.txt
   "Para Cursar.txt" "Para Final.txt"
```

Ejemplo 2.
```
SimularSsl 2005 5.txt LPC.txt
   LPREF.txt > Salidas.txt
```
En este último ejemplo la salida standard es redireccionada al archivo "Salidas.txt" mediante el símbolo mayor (">") perteneciente al lenguaje de la mayoría del módulo intérprete de comandos de la mayoría de los sistemas operativos (e.g.: cmd.exe).

Ejemplo 3.
```
SimularSsl
   Descripción del programa comando.
   Sintaxis del progrmama comando.
```

Ejemplo 4.
```
SimularSsl dosmilcinco 5.txt LPC.txt LPREF.txt
   Año inválido
```

Ejemplo 5.
```
SimularSsl 2001 Calendario.txt "Para Cursar.txt"
   Cantidad insuficiente de argumentos
```

Ejemplo 6.
```
SimularSsl 2005 CA.txt CU.txt FI.txt D.txt
   Cantidad extra de argumentos
```

Implementación

El programa comando se implementará en un sólo archivo fuente ANSI C. El programa hará uso de la biblioteca standard y de las bibliotecas que implementan los TADs Actividad, CalendarioAcademico, MesaDeExamenFinal y Curso.

Para su compilación y enlace, en BCC32 se deberá utilizar el siguiente comando: `BCC32 SimularSsl.c Actividad.lib Curso.lib CalendarioAcademico.lib MesaDeExamenFinal.lib`

En el grupo Yahoo de la cátedra estarán disponibles los tres archivos de texto: **CalendarioAcademico**.txt, **ParaCursar**.txt y **ParaFinal**.txt.

Pseudocódigo – Guía para la Construcción del Programa de Simulación

El diseño del programa se puede describir de forma resumida como: una sentencia de iteración que recorre el calendario académico con dos sentencias:
1. Una sentencia que despliega el detalle del evento que se está atendiendo, y
2. Una sentencia de selección que delega la atención de cada uno de los eventos del calendario como veremos adelante.

♦A continuación se presenta el pseudocódigo y guía de implementación para la función `main`. Luego se presenta la estructura preliminar del programa (incluyendo un mayor detalle para la función `main`), y por último, el pseudocódigo y guía de implementación para el resto de las funciones.

1. main

La función `main` tiene como responsabilidad verificar los argumentos de la línea de comando, leer el calendario académico a memoria, recorrerlo e ir delegando la atención de cada evento a las funciones especializadas.

```
int main(int argc, char* argv[]){
   Verificar los argumentos;
   Leer el calendario académico;
   Recorrer el calendario académico;
      Delegar la atención del evento;
}
```

Estructura Preliminar del Programa

A continuación se detalla un poco más la función `main` y se la presenta en el contexto de la estructura general del programa.
La inscripción al primer cuatrimestre es al final del año académico anterior, por lo que ese evento es disparado incondicionalmente antes de comenzar el ciclo de simulación. Luego, según el calendario académico, es disparado una vez más antes de comenzar el segundo cuatrimestre.

```
/* SimularSsl.c */

typedef enum {
   DemasiadosArgumentos,
   PocosArgumentos,
   AnioInvalido,
   NoSePudoAbrirArchivo,
   NoSePudoCerrarArchivo,
   NoSePudoLeerDeArchivo,
   NoSePudoEscrbrirEnArchivo, ... , etc.
} Error;

void MostrarError(Error unError);
void MostrarAyuda(void);
/* Función que implementa un AF reconocedor
 * ver Capítulo 11 de [MUCH200]
 */
int  EsAnio( const char* unaCadena );
...

   declaración del resto de las funciones;
   ...

   #include "Actividad.h"
   #include "CalendarioAcademico.h"
   #include "MesaDeExamenFinal.h"
   #include "Curso.h"
   #include <stdio.h>

   int main( int argc, char* argv[] ){

      if( ! EsAnio( argv[1] ) ){
         MostrarError( AnioInvalido );
         return EXIT_FAILURE;
      }

      ...
      Verificar el resto de los argumentos;
      ...
```

```
{ /* comienzo del bloque principal del programa comando */
   int elCuatrimestre = 1;
   int elNumeroDeMesa = 1;
   int elNumeroDeClase = 1;
   const char* losCursos[2][2] = {
      {"K2001", "Amelia Mesa"  }, /* 1er cuatrimestre */
      {"K2002", "Rodolfo Fleco"}  /* 2do cuatrimestre */
   };
   const char* lasMesas[10][3] = {
      {"Rodolfo Fleco", "Miguel Puente", "Celina Flores"},
      {"Celina Flores", "Amelia Mesa"  , "Rodolfo Fleco"},
      {"Celina Flores", "Rodolfo Fleco", "Amelia Mesa"  },
      {"Miguel Puente", "Amelia Mesa"  , "Celina Flores"},
      {"Celina Flores", "Amelia Mesa"  , "Miguel Puente"},
      {"Amelia Mesa"  , "Rodolfo Fleco", "Celina Flores"},
      {"Rodolfo Fleco", "Celina Flores", "Amelia Mesa"  },
      {"Amelia Mesa"  , "Celina Flores", "Miguel Puente"},
      {"Celina Flores", "Rodolfo Fleco", "Miguel Puente"},
      {"Amelia Mesa"  , "Miguel Puente", "Rodolfo Fleco"}
   };
   CalendarioAcademico* elCalendario = LeerCalendarioAcademico(nombreArchivoCalendario, elAnio);

   Curso unCurso = AtenderEventoInscripcionACurso(
      nombreArchivoParaCursar,
      losCursos[elCuatrimestre-1][0],
      elAnio,
      elCuatrimestre,
      losCursos[elCuatrimestre-1][1]
   );

   /* Simular, recorriendo el calendario y atendiendo los eventos */
   while( elCalendario tenga más actividades, avanzar ){

      /* Obtener y mostrar la actividad actual */
      laActividad = CalendarioAcademico_GetActividadActual(elCalendario);
      MostrarActividad(&laActividad);

      /* Determinar el evento y delegar su atención */
      switch( Actividad_GetTipoDeActividad(laActividad) ){

         /* Eventos de un curso */

         case PrimeraClase:
         case ClaseNormal:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            break;

         case EvaluacionIndividual:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            Curso_TomarEvaluacionIndividual(&unCurso);
            break;

         case EvaluacionGrupalEscrita:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            Curso_TomarEvaluacionGrupalEscrita(&unCurso);
            break;

         case EvaluacionGrupalOral:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            Curso_TomarEvaluacionGrupalOral(&unCurso);
            break;

         case PrimerRecuperatorio:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            Curso_TomarPrimerRecuperatorio(&unCurso);
            break;

         case SegundoRecuperatorio:
            elNumeroDeClase++;
            Curso_TomarAsistencia(&unCurso);
            Curso_TomarSegundoRecuperatorio(&unCurso);
            break;

         case UltimaClase:
            elNumeroDeClase++;
            AtenderEventoUltimaClase(
               nombreArchivoParaCursar
               nombreArchivoParaFinal,
               &unCurso,
               Actividad_GetDia(laActividad),
               Actividad_GetNumeroMes(laActividad),
            );
            elCuatrimestre++;
            break;
```

```
            case InscripcionACurso:
               /* Para que una variable del tipo curso pueda contener
                  un valor diferente que el valor dado en la inicialización,
                  el campo codigo no debe tener el calificador const */
               unCurso = AtenderEventoInscripcionACurso(
                  nombreArchivoParaCursar,
                  losCursos[elCuatrimestre-1][0],
                  elAnio,
                  elCuatrimestre,
                  losCursos[elCuatrimestre-1][1]
               );
               break;


            /* Eventos de una mesa de examen final */

            case InscripcionAExamenFinal:
               unaMesa = AtenderEventoInscripcionAExamenFinal(
                  nombreArchivoParaFinal,
                  Actividad_GetDia(laActividad),
                  Actividad_GetNumeroMes(laActividad),
                  lasMesas[elNumeroDeMesa - 1][0],
                  lasMesas[elNumeroDeMesa - 1][1],
                  lasMesas[elNumeroDeMesa - 1][2]
               );
               elNumeroDeMesa++;
               break;

            case ExamenFinal:
               AtenderEventoExamenFinal(
                  nombreArchivoParaFinal, nombreArchivoParaCursar, &unaMesa, elNumeroDeMesa
               );
               break;
         } /* fin switch */
      } /* fin while */

      Destruir elCalendario;
   } /* fin del bloque principal del programa comando */

   return EXIT_SUCCESS;
}

...
Definición del resto de las funciones;
...
```

2. LeerCalendarioAcedemico

Proceso de Secretaría Académica: **Llenar Calendario Académico**. Lee las actividades desde el archivo dado, las agrega a elCalendarioAcademico y lo retorna. El archivo contiene líneas, con una actividad en cada línea. Las líneas deben ser leídas con `fgets` y *tokenizadas* con `strtok`.

```
CalendarioAcademico* LeerCalendarioAcademico(
   const char* nombreArchivoCalendario,
   int         unAnio
){
   Abrir archivo;
   elCalendario = CalendarioAcademico_Crear(unAnio);
   while( se pueda leer una línea con fgets ){
      separar la línea en 4 tokens
         mediante 4 invocaciones a strtok;
      Agregar la actividad a elCalendario;
   }
   Cerrar archivo;
   return elCalendario;
}
```

3. AtenderEventoInscripcionACurso

Proceso de Secretaría Académica: **Inscribir a Curso**. Las estadísticas indican que el **60%** de los alumnos en condiciones de inscribirse a un curso realizan su inscripción, simular esta situación con la función `rand`. El archivo contiene líneas con un legajo cada una; las líneas deben ser leídas con la función `fscanf`. Utilizando las operaciones del TAD Curso, la función retorna un valor del TAD Curso con su conjunto de *alumnos* completo. Luego de invocar a esta función, el archivo no contendrá las líneas con los legajos que sí fueron inscriptos.

```
Curso AtenderEventoInscripcionACurso(
   const char* nombreArchivoParaCursar,
   const char* unCodigo,
   int         unAnio,
   int         unCuatrimestre,
   const char* unProfesor,
){
   Abrir archivo;
   Crear archivo temporal;
   Crear nuevo archivo;

   n = 0;
   Leer con fscanf("%s") del archivo{
      Mediante rand():
         el 60% {
            pasa al archivo temporal
            con fprintf("%s\n");
            n++;
         }
         el otro 40% pasa al nuevo archivo
            con fprintf("%s\n");
   }
   Cerrar archivos;

   elCurso = Curso_Crear(
      unCodigo, unAnio, unCuatrimestre, unProfesor, n
   );

   Leer legajos del archivo temporal con
      fscanf("%s") {
         inscribir a elCurso;
   }

   Eliminar archivo temporal;
   Renombrar el nuevo archivo;

   return elCurso;
}
```

4. MostrarActividad

Despliega en `stdout` un separador y un mensaje descriptivo sobre la actividad que se está atendiendo.

```
void MostrarActividad(
   const Actividad* unaActividad
){
   Escribir separador de 40 guiones;
   Escribir día, mes largo, tipo de actividad
      y descripción de unaActividad;
}
```

5. AtenderUltimaClase

Toma asistencia por última vez, cierra el curso, emite acta en formato texto y binario, emite regularizados y recursantes. Los nombres de los archivos son: "Acta**CAC**.txt", "ActaBinario**CAC**.bin", "Regularizados**CAC**.txt" y "Recursantes**CAC**.txt". Siendo el sufijo CAC la concatenación del código, año y cuatrimestre del curso. Libera los recursos tomados por elCurso. Delega la distribución de los regularizados y de los recursantes a dos funciones especializadas.

```
void AtenderEventoUltimaClase(
   const char* nombreArchivoParaCursar,
   const char* nombreArchivoParaFinal,
   Curso*      unCurso,
   int         unDia,
   int         unMes
){
   Curso_TomarAsistencia(unCurso);
   Curso_Cerrar(unCurso, unDia, unMes);

   elSufijo =
      Curso_GetCodigo(unCurso) +
      Curso_GetAnio(unCurso)   +
      Curso_GetCuatrimestre(unCurso);
   Crear los cuatro archivos
   Curso_EmitirActaTexto(unCurso, archivo1);
   Curso_EmitirActaBinario(unCurso, archivo2);
   Curso_EmitirRegularizadosTexto(
      unCurso, archivo3
   );
   Curso_EmitirRecursantesTexto(unCurso, archivo4);
   Cerrar archivos;

   Curso_Destruir(unCurso);

   RecolectarRecursantes(
      archivo3, nombreArchivoParaCursar
   );
   RecolectarRegularizados(
      archivo3, nombreArchivoParaFinal
   );
}
```

5.1. RecolectarRegularizados

Proceso de Secretaría Académica: **Recolectar regularizados**. Toma los legajos del archivo con los regularizados y los agrega al archivo con los legajos de los alumnos listos para rendir examen final, con cantidad de veces rendida igual a cero. El archivo contiene líneas con un legajo y una fecha; las líneas deben ser leídas con la función `fgets` y el legajo será obtenido con una invocación a `strtok`. Se utilizará `fprintf` para escribir el archivo destino.

```
void RecolectarRegularizados(
   const char* nombreArchivoRegularizados,
   const char* nombreArchivoParaFinal
){
   Abrir archivos;
   Por cada legajo regularizado leído con fgets{
      invocar a strtok para obtener legajo;
      fprintf(destino, "%s 0\n", legajo);
   }
   Cerrar archivos;
}
```

5.2. RecolectarRecursantes

Proceso de Secretaría Académica: **Recolectar recursantes**. Toma los legajos del archivo con los recursantes y los agrega al archivo con los legajos de los alumnos listos para cursar. El archivo contiene líneas con un legajo cada una; las líneas deben ser leídas con la función `fscanf`. Se utilizará `fprintf` para escribir el legajo en el archivo destino.

```
void RecolectarRecursantes(
   const char* nombreArchivoRecursantes,
   const char* nombreArchivoParaCursar
){
   Abrir archivo recursantes para lectura;
   Abrir archivo destino en modo append;

   Por cada legajo recursante
   leído con fscanf("%s"){
      fprintf(destino, "%s\n", legajo);
   }
   Cerrar archivos;
}
```

6. AtenderEventoInscripcionAExamenFinal

Proceso de Secretaría Académica: **Inscribir a Mesa de Examen Final**. Las estadísticas indican que el **75%** de los alumnos en condiciones de rendir examen final realizan su inscripción, simular esta situación con la función `rand`. El archivo contiene líneas con un legajo y con la cantidad de veces que ese alumno rindió final. Las líneas deben ser leídas con `fscanf`. La función retorna una MesaDeExamenFinal con su conjunto de *inscriptos* completo.

```
MesaDeExamenFinal
   AtenderEventoInscripcionAExamenFinal(
   const char* nombreArchivoParaFinal,
   int         unDia,
   int         unMes,
   const char* unPresidente,
   const char* unPrimerVocal,
   const char* unSegundoVocal
){
   Abrir archivo;
   Crear archivo temporal;

   n = 0;
   Leer con fscanf("%s %d") del archivo{
      Mediante rand() el 75%{
         pasa al archivo temporal con
         fprintf("%s\n"); n++;
      }
   }

   laMesa = Mesa_Crear(
      unDia, unMes, unPresidente,
      unPrimerVocal, unSegundoVacal, n
   );

   Leer con fscanf("%s") del archivo temporal{
      inscribir a laMesa;
   }

   Cerrar archivos;
   Eliminar archivo temporal;

   return laMesa;
}
```

7. AtenderEventoExamenFinal

Toma asistencia, evalúa, emite el acta en formato texto, los ausentes, los aprobados y los desaprobados. Los nombres de los archivos son: "`ActaN.txt`", "`AusentesN.txt`", "`AprobadosN.txt`" y "`DesaprobadosN.txt`". Siendo el sufijo `N` el número de mesa de final. Libera los recursos tomados por elCurso. Delega la distribución de los ausentes y de los desaprobados a dos funciones especializadas.

```
void AtenderEventoExamenFinal(
   const char*        nombreArchivoParaCursar,
   const char*        nombreArchivoParaFinal,
   MesaDeExamenFinal* unaMesa,
   int                unNumeroDeMesa
){
   MesaDeExamenFinal_TomarAsistencia(unaMesa);
   MesaDeExamenFinal_Evaluar(unaMesa);

   Crear los cuatro archivos con el sufijo;
   MesaDeExamenFinal_EmitirActaTexto(
      unaMesa, archivo1
   );
   MesaDeExamenFinal_EmitirAprobadosTexto(
      unaMesa, archivo4
   );
   MesaDeExamenFinal_EmitirAusentesTexto(
      unaMesa, archivo3
   );
   MesaDeExamenFinal_EmitirDesaprobadosTexto(
      unaMesa, archivo4
   );
   Cerrar archivos;

   MesaDeExamenFinal_Destruir(unaMeasa);

   RemoverAprobados(
      archivo2, nombreArchivoParaFinal
   );
   DistribuirDesaprobados(
      archivo4,
      nombreArchivoParaCursar,
      nombreArchivoParaFinal
   );
}
```

7.1. RemoverAprobados

Proceso de Secretaría Académica: **Remover Aprobados**. Remueve del archivo de alumnos listos para rendir final los alumnos que hayan aprobados. El archivo contiene líneas con un legajo y una fecha. El archivo con alumnos aprobados contiene un legajo y una nota por línea. Ambos archivos deben ser leídos con `fscanf`. Se utilizará `fprintf` para escribir el archivo destino.

```
void RemoverAprobados(
   const char* nombreArchivoRegularizados,
   const char* nombreArchivoParaFinal,
   ...
){
   Analizar el problema;
   Diseñar posibles soluciones;
   Seleccionar una y descrbir su estrategia;
}
```

7.2. DistribuirDesaprobados

Proceso de Secretaría Académica: **Distribuir Desaprobados**. Incrementa la cantidad de veces que un alumno rindió final, ó lo remueve del archivo listos para rendir final y lo envía como recursante al archivo listos para cursar. Toma los legajos del archivo de texto con los desaprobados, cada línea contiene un legajo y un entero que representa la nota; las líneas deben ser leídas con la función `fscanf`. Se utilizará `fprintf` para escribir en los diferentes archivos.

```
void AtenderEventoUltimaClase(
   const char* nombreArchivoRegularizados,
   const char* nombreArchivoParaCursar,
   const char* nombreArchivoParaFinal,
   ...
){
   Analizar el problema;
   Diseñar posibles soluciones;
   Seleccionar una y descrbir su estrategia;
}
```

Conceptos de ANSI C necesarios

- Los necesarios para las entregas anteriores.
- Programa comando: `int main(int argc, char* argv[])`.
- Función: `exit`.
- Funciones de Entrada/Salida Standard: `fprintf`, `fgets`, `fscanf`, `fopen`, `fclose` y `ferror`.
- Función para *tokenizar* (separar en *tokens*) cadenas: `strtok`.

Otras Restricciones

- Al salir de un ciclo de lectura de un archivo invocar a la función **`ferror`**. Las funciones de lectura retornan una valor, como se describe en [M3] indicando si se la lectura fue exitosa. Además, en caso de que la lectura no haya sido exitosa, colocan una señal en la estructura **FILE**; esta señal indica porque no se pudo realizar la lectura: 1. porque se alcanzó el fin de archivo, ó 2. porque hubo algún error de lectura. Para determinar el porqué se hace uso de las funciones **`ferror`** y **`feof`**.
- Ante alguna excepción en la ejecución de la simulación, se invocará a la función **`MostrarError`** con el argumento correspondiente y seguidamente a la función **`exit`** con `EXIT_FAILURE` cómo argumento (i.e.: {**`MostrarError`**`(e);` **`exit`**`(EXIT_FAILURE);}`).
- La implementación de la biblioteca y el programa de simulación deben utilizar el lenguaje y los elementos de la biblioteca estándar especificados en ANSI C [M3].
- Luego de que la biblioteca, la aplicación de prueba y el programa de simulación hayan sido construidos con la herramienta de desarrollo elegida por el equipo, deben ser verificados mediante la herramienta de desarrollo "Borland C++ Compiler 5.5 with Command Line Tools" (BCC32) configurada tal como dicta la cátedra [TP #0].
- Los procesos de compilación con BCC32 no deben emitir warnings (mensajes de advertencia) ni, por supuesto, mensajes de error.
- Los identificadores de las funciones que implementan las operaciones, de las variables, tipos y demás elementos así como los nombres de los archivos deben ser los indicados a lo largo de este enunciado.
- Cualquier decisión que el equipo tome sobre algún punto no aclarado en el enunciado debe ser agregada como hipótesis de trabajo.

Sobre la Entrega

Tiempo

- Catorce (14) días después de analizado el enunciado de la primera parte de esta entrega en clase.

Forma

El trabajo debe presentarse en hojas **A4 abrochadas** en la esquina superior izquierda. En el encabezado de cada hoja debe figurar el **título** del trabajo, el título de **entrega**, el código de **curso** y los **apellidos** de los integrantes del equipo. Las hojas deben estar enumeradas en el pie de las mismas con el formato "**Hoja n de m**".

El código fuente de cada componente del TP debe comenzar con un comentario encabezado, con todos los datos del equipo de trabajo: **curso**; **legajo**, **apellido** y **nombre** de cada integrante del equipo y **fecha de última modificación**. La fuente a utilizar en la impresión debe ser una fuente de ancho fijo (e.g. `Courier new`, `Lucida Console`).

1. TAD CalendarioAcademico

1.1. **Especificación**. Especificación completa, extensa y sin ambigüedades de los valores y de las operaciones del TAD.
1.2. **Implementación**
1.2.2. Biblioteca que implementa el TAD
1.2.2.1. Listado de código fuente del archivo encabezado, **parte pública**, **CalendarioAcademico.h.**
1.2.2.2. Listado de código fuente del la definición de la Biblioteca, **parte privada**, **CalendarioAcademico.c.** Las funciones privadas deberán ser precedidas por su documentación, un comentario con la documentación de la función indicando, entre otras cosas, propósito de la función, semántica de los parámetros *in*, *out* e *inout*.
1.2.3. **Salidas**. Captura impresa de la salida del proceso de traducción (BCC32 y TLIB). Utilizar fuente de ancho fijo.

2. Aplicación de Prueba

2.1. **Código Fuente**. Listado del código fuente de la aplicación de prueba, **CalendarioAcademicoAplicacion.c.**
2.2. **Salidas**
2.2.1. Captura impresa de la salida del proceso de traducción (BCC32). Utilizar fuente de ancho fijo.
2.2.2. Captura impresa de las salidas de la aplicación de prueba. Utilizar fuente de ancho fijo.

3. Programa de Simulación.

3.1. **Código Fuente**. Listado del código fuente del programa de aplicación, **SimulacionAnioAcademicoSsl.c.**
3.2. **Salidas**
3.2.1. Captura impresa de la salida del proceso de traducción (BCC32). Utilizar fuente de ancho fijo.
3.2.2. Captura impresa de las salidas del programa de aplicación. Utilizar fuente de ancho fijo.

4. Copia Digitalizada

CD ó disquette con copia de solamente los 4 archivos de código fuente (CalendarioAcademico.h, CalendarioAcademico.c, CalendarioAcademicoAplicacion.c y SimulacionAnioAcademicoSsl.c).
no se debe entregar ningún otro archivo.

5. Formulario de Seguimiento de Equipo.

Sobre la Entrega Final

Tiempo

- Al momento de dar el Coloquio.

Forma

Al momento de dar el Coloquio, el integrante del equipo debe tener disponible una carpeta que reúna las cuatro entregas, según las normas de presentación de cada una de esas cuatro entregas, incluyendo la presente.