En este documento se describen dos archivos de 'cabecera' (**funciones.h** e **info.h**) y sus correspondientes desarrollos (**funciones.c** e **info.c**) .
Se entiende que en **info.c** se desarrolla el ingreso y validaciones variadas de los campos de información para el tipo de dato requerido en particular, en tanto que en **funciones.h** se desarrollan funciones de uso más general .
Se destaca que, tal vez, el tipo de dato correspondiente a la fecha debería haberse declarado en **funciones.h**, pero se ha optado por declararlo en **info.h** .
Se ha querido ejemplificar el ingreso, validación y hasta en algunos casos (p.ej.: para el apellido y nombre y para el sexo), corregir lo ingresado .

Archivo : funciones.h

```
#ifndef FUNCIONES_H__
#define FUNCIONES_H__

#include "info.h"


char cMenu(const char *mensaje, const char *opciones);

int nEsBisiesto(int an);

int nFechaValida(const t_fecha *f);

#endif
```

Archivo : funciones.c

```
#include <stdio.h>
#include <string.h>

#include "funciones.h"


/*****************************************************************************
 * función        :  cMenu
 * recibe         :  char *mensaje  - mensaje a mostrar
 *                   char *opciones - opciones válidas a ingresar
 * devuelve       :  un carácter válido (es uno de los recibidos en el 2do arg)
 * acción         :  muestra el mensaje e ingresa una opción, repitiendo hasta
 *                      que la opción ingresada sea válida,
 * observaciones  :
 */
char cMenu(const char *mensaje, const char *opciones)
{
   char           aux;

   do
   {
      printf("%s", mensaje);
      fflush(stdin);
      scanf("%c", &aux);
   } while(!strchr(opciones, aux));
   return aux;
}


/*****************************************************************************
 * función        :  nFechaValida
 * recibe         :  t_fecha *f  - puntero a la fecha a validar
 * devuelve       :  1 (verdad) - si es una fecha válida
 *                   0 (falso)  - si no lo es
 * acción         :  se vale de un array bidimensional preinicializado
 *                      con el máximo de días para los años no bisiestos y
```

Archivo : funciones.c

```
 *                      bisiestos
 * observaciones  :  adaptado del libro de Kernigan y Ritchie
 *                   responde al calendario Juliano, adoptado por el papa
 *                      Gregorio XIII, con validez desde 1582 hasta dentro de
 *                      unos 3500 años
 */
int nFechaValida(const t_fecha *fec)
{
   static char    dias[][12]   =
                     { { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                       { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} };

   return fec->me >=    1 && fec->me <= 12    &&
          fec->an >= 1592 && fec->an <= 2100  &&
          fec->di >     0 &&
          fec->di <= dias[nEsBisiesto(fec->an)][fec->me - 1];
}


/*****************************************************************************
 * función        :  nEsBisiesto
 * recibe         :  int an   - año sobre el que se determina si lo es o no
 * devuelve       :  1 (verdad) - si es bisiesto
 *                   0 (falso)  - si no lo es
 * acción         :  los años bisiestos son (para el calendario Greghoriano)
 *                      los que son múltiplos de 4 pero no de 100, salvo que
 *                      sean múltiplos de 400
 * observaciones  :  adaptado del libro de Kernigan y Ritchie
 *                   responde al calendario Juliano, adoptado por el papa
 *                      Gregorio XIII, con validez desde 1582 hasta dentro de
 *                      unos 3500 años
 */
int nEsBisiesto(int an)
{
   return an % 4 == 0 && an % 100 != 0 || an % 400 == 0;
}
```

Archivo : info.h

```
#ifndef INFO_H__
#define INFO_H__


/*
 * declaración de los tipos de datos para la información :
 *    1ro.  para la fecha
 *    2do.  para la información a cargar (que incluye una fecha)
 */
typedef struct
{
   int di;
   int me;
   int an;
} t_fecha;

typedef struct
{
   long     dni;
   char     apyn[36];
   char     sex;
   t_fecha  fec;
   double   importe;
} t_info;
```

Archivo : info.h

```
/*
 * funciones para el manejo de la información
 */
int ingresar(t_info *d);
void mostrar(const t_info *d);


#endif
```

Archivo : info.c

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "info.h"
#include "funciones.h"

void eliminar_caracter(char *cad, char letra);
void arreglar_apyn(char *s, size_t tam);
int apyn_valido(char *s);
int importe_valido(double valor);


/*****************************************************************************
 *** funciones para el manejo de la información ***
 */

/*****************************************************************************
 * función        :  ingresar
 * recibe         :  t_info *d    - dirección de memoria en que almacena la
 *                                  información
 * devuelve       :  valor booleano indicador de que se ingresó o no
 * acción         :  mediante mensajes de orientación procede al ingreso de
 *                      la info en cuestión
 * observaciones  :
 */
int ingresar(t_info *d)
{
   do
   {
      d->dni = 0L;
      printf("(para terminar ingrese un DNI negativo)\n"
             "D. N. I.           : ");
      fflush(stdin);
      scanf("%ld", &d->dni);
      if(d->dni < 0L)
         return 0;
   } while(d->dni <  1000000L || d->dni >   40000000L &&
           d->dni < 90000000L || d->dni >= 100000000L);

   do
   {
      printf("Apellido y nombre  : ");
      fflush(stdin);
      fgets(d->apyn, sizeof(d->apyn), stdin);
      eliminar_caracter(d->apyn, '\n');
      arreglar_apyn(d->apyn, sizeof(d->apyn));
   } while(!apyn_valido(d->apyn));

/* el siguiente código se reemplaza por el uso de la función de menú
   do
   {
```

Archivo : info.c

```
      printf("Sexo (M/F)         : ");
      fflush(stdin);
      scanf("%c", &d->sex);
      d->sex = toupper(d->sex);
   } while(d->sex != 'M' && d->sex != 'F');
 */
   d->sex = toupper(cMenu("Sexo (M/F)         : ", "MmFf"));

   do
   {
      d->fec.di = d->fec.me = d->fec.an = -1;
      printf("Fecha (dd/mm/aaaa) : ");
      fflush(stdin);
      scanf("%d/%d/%d", &d->fec.di, &d->fec.me, &d->fec.an);
   } while(!nFechaValida(&d->fec));

   do
   {
      d->importe = .001;
      printf("Importe            : ");
      fflush(stdin);
      scanf("%lf", &d->importe);
   } while(!importe_valido(d->importe));
   return 1;
}


/*****************************************************************************
 * función        :  arreglar_apyn
 * recibe         :  char  *s   - puntero al apellido y nombres a validar
 *                   size_t tam - sizeof del array de char original
 * devuelve       :  n/a
 * acción         :  "arregla" el contenido de la cadena de carácteres :
 *                      cambiando tabulaciones u otros carácteres de control
 *                         por blanco,
 *                      eliminando el exceso de espacios en blanco,
 *                      forzando que haya un blanco después de la coma
 *                      dejando en mayúscula el primer carácter de cada
 *                         palabra y en minúsculas los demás
 *                      etc
 * observaciones  :  INVOCAR ANTES QUE A  apyn_valido
 */
void arreglar_apyn(char *s, size_t tam)
{
   char          *aux;

   /* cambio de cualquier carácter de control por ' ' */
   aux = s;
   while(*aux)
   {
      if(iscntrl(*aux))
         *aux = ' ';
      aux++;
   }

   /* eliminar el exceso de blancos */
   aux = s;
   while((aux = strstr(aux, "  ")) != NULL)
      strcpy(aux, aux + 1);

   /* eliminar blanco antes de la primer palabra */
   if(*s == ' ')
      strcpy(s, s + 1);

   /* eliminar blanco antes de la coma */
   if((aux = strstr(s, " ,")) != NULL)
```

Archivo : info.c

```
      strcpy(aux, aux + 1);

   /* eliminar blanco después de la última palabra */
   if((aux = strchr(s, '\0')) != NULL && aux > s && *(aux - 1) == ' ')
      *(aux - 1) = '\0';

   /* eliminar blanco antes y después de apóstrofos */
   aux = s;
   while((aux = strchr(aux, '\'')) != NULL)
   {
      if(*(aux + 1) == ' ')
         strcpy(aux + 1, aux + 2);
      if(aux > s && *(aux - 1) == ' ')
         strcpy(aux - 1, aux);
      if(*aux == '\'')
         aux++;
   }

   /*
    * forzar blanco después de la coma :
    *    ADVIERTA que :  esto extiende la cadena, por lo cual la función
    *       recibe el tamaño o sizeof de la cadena apuntada
    */
   if((aux = strchr(s, ',')) != NULL && *(aux + 1) != ' ')
   {
      memmove(aux + 1, aux, tam - 1 > strlen(s) ? strlen(aux) : strlen(aux + 1));
      *(aux + 1) = ' ';
   }

   /*
    * forzar a mayúscula la primer letra y a minúscula
    *    las siguientes de cada palabra
    */
   aux = s + 1;
   *s = toupper(*s);
   while(*aux)
   {
      if(*(aux - 1) == ' ' || *(aux - 1) == '\'')
         *aux = toupper(*aux);
      else
         *aux = tolower(*aux);
      aux++;
   }

}

/*****************************************************************************
 * función        :  apyn_valido
 * recibe         :  char  *s    - puntero al apellido y nombres a validar
 * devuelve       :  valor booleano indicador de que es válida o no
 * acción         :  válida que :
 *                      haya una y sálo una coma(,) y que
 *                      el apellido tenga al menos dos carácteres y el nombre
 *                         al menos tres
 * observaciones  :
 */
int apyn_valido(char *s)
{
   char          *aux;

   /* si no hay una sóla coma(,) ya no es válido */
   if((aux = strchr(s, ',')) == NULL)
      return 0;
   else
      if((aux = strchr(aux + 1, ',')) != NULL)
         return 0;
```

Archivo : info.c

```
   /* validación, apellido al menos dos carácteres, y nombre al menos tres */
   if((int)((aux = strchr(s, ',')) - s) < 2)
      return 0;
   if((int)(strchr(aux, '\0') - aux) < 4)
      return 0;

   return 1;
}


/*************************************************************************
 * función        :  eliminar_caracter
 * recibe         :  char *cad   -cadena de carácteres
 * devuelve       :  char letra  -carácter a eliminar
 * acción         :  valiéndose de un puntero auxiliar(aux), busca (empleando
 *                      strchr) todas las ocurrencias del caracter(letra),
 *                      si lo encuentra, copia el resto de la cadena 'pisando'
 *                      el carácter
 * observaciones  :
 */
void eliminar_caracter(char *cad, char letra)
{
   char          *aux          = cad;

   while((aux = strchr(aux, letra)) != NULL)
      strcpy(aux, aux + 1);
}
/*  otra versión sin funciones de biblioteca   */
/*
{
   char *aux = cad;

   while((*aux = *cad++) != '\0')
      if(*aux != letra)
         aux++;
}
*/


/*************************************************************************
 * función        :  importe_valido
 * recibe         :  const float valor    -flotante a validar
 * devuelve       :  valor booleano
 * acción         :  determina (si hay parte decimal) que no exceda los tres
 *                      dígitos
 * observaciones  :
 */
int importe_valido(double valor)
{
   char           texto[30];
   double         aux;

   /* si no tiene (como máximo) dos decimales */
   sprintf(texto, "%.2lf", valor);
   sscanf(texto, "%lf", &aux);
   if(aux != valor)
      return 0;
   return 1;
}

/*************************************************************************
 * función        :  void mostrar
 * recibe         :  const t_info *d   -dirección de memoria de la info
 *                                        a mostrar
 * devuelve       :  n/a
```

Archivo : info.c

```
 * acción         :  muestra la información cuya dirección de memoria recibe
 * observaciones  :
 */
void mostrar(const t_info *d)
{
   printf("D. N. I. :  %ld\n"
          "Apellido y Nombre :  %s\n"
          "Sexo              :  %c\n"
          "Fecha             :  %02d/%02d/%04d\n"
          "Importe           :  %7.2lf\n\n",
          d->dni,
          d->apyn,
          d->sex,
          d->fec.di, d->fec.me, d->fec.an,
          d->importe);
}


/*****************************************************************************
 * función        :  nada
 * recibe         :  n/a
 * devuelve       :  n/a
 * acción         :  obliga a que al compilar un programa, se incluyan las
 *                      bibliotecas de formatos de punto flotante
 * observaciones  :  A ESTA FUNCION NO ES NECESARIO INVOCARLA
 *                   en los compiladores de Borland cuando se hace scanf para
 *                      ingresar variables en punto flotante, el compilador
 *                      puede no haber incluido los formatos de punto flotante
 *                   esta función es una alternativa para hacer que el
 *                      compilador las incluya
 *                   de todos modos en la documentación del compilador se
 *                      indica emplear lo siguiente :
 *                      extern void _floatconvert();
 *                      #pragma extref _floatconvert
 *
 *                   en el caso de los compiladores de Microsoft sucede lo
 *                      mismo, en su documentación se indica que basta
 *                      con inicializar cualquier variable en punto flotante
 *                      con cualquier valor, en cualquier lugar del programa
 */
void nada(void)
{
   float          f            = 0.0f,
                 *p;

   p = &f;
   f = *p;
}
```

En este documento se describen dos archivos de 'cabecera' (**pila.h** y **cola.h**) y sus correspondientes desarrollos (**pila.h** y **cola.h**), para las implementaciones estáticas de ambas estructuras de datos .

Archivo : pila.h – (implementación estática)

```
#ifndef PILA_H__
#define PILA_H__

#include "info.h"



/*****************************************************************************
 * declaración de la máxima cantidad de elementos a cargar en la pila
 */
#define MAX_PILA        30

/*****************************************************************************
 * declaración del tipo de dato para el almacenamiento (en un array) de los
 *    elementos de información a cargar en la pila
 */
typedef struct
{
   t_info   pila[MAX_PILA];
   int      tope;
} t_pila;



/*****************************************************************************
 * primitivas para el manejo de pilas
 */
void vCrearPila(t_pila *p);

void vVaciarPila(t_pila *p);

int nPilaLlena(const t_pila *p);

int nPilaVacia(const t_pila *p);

int nPonerEnPila(t_pila *p, const t_info *d);

int nSacarDePila(t_pila *p, t_info *d);

int nVerTopePila(t_pila *p, t_info *d);


#endif
```

Archivo : cola.h – (implementación estática)

```
#ifndef COLA_H__
#define COLA_H__

#include "info.h"



#define MAX_COLA        30

typedef struct
{
   t_info cola[MAX_COLA];
   int pri,
       ult;
} t_cola;
```

Archivo : cola.h – (implementación estática)

```
/*****************************************************************************
 * primitivas para el manejo de colas
 */
void vCrearCola(t_cola *p);

void vVaciarCola(t_cola *p);

int nColaLlena(const t_cola *p);

int nColaVacia(const t_cola *p);

int nPonerEnCola(t_cola *p, const t_info *d);

int nSacarDeCola(t_cola *p, t_info *d);

int nVerPrimeroCola(t_cola *p, t_info *d);


#endif
```

Archivo : pila.c – (implementación estática)

```
#include "pila.h"



#define CREARPILA(X)    ( (X)->tope =  0 )


#define VACIARPILA      CREARPILA


#define PILALLENA(X)    ( (X)->tope == MAX_PILA )


#define PILAVACIA(X)    ( (X)->tope == 0 )


/*****************************************************************************
 * función        :  vCrearPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  n/a
 * acción         :  inicializa el tope para la posicion 0 (próxima a utilizar)
 * observaciones  :  - esta función coincide con la de Vaciar la Pila y su
 *                     código se repite en otras primitivas
 */
void vCrearPila(t_pila *p)
{
   p->tope = 0;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* CREARPILA(p);                                                           */
}


/*****************************************************************************
 * función        :  nPilaLlena
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  1 (verdad)  si está llena
 *                   0 (falso)   si no lo está
 * acción         :  determina si el miembro tope de la pila contiene el
 *                     máximo posible
 * observaciones  :  - el código de esta función se utiliza en otras funciones
 */
```

Archivo : pila.c – (implementación estática)

```
int nPilaLlena(const t_pila *p)
{
   return p->tope == MAX_PILA;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* return PILALLENA(p);                                                    */
}


/*************************************************************************
 * función        :  nPilaVacia
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  1 (verdad)  si está vacía
 *                   0 (falso)   si no lo está
 * acción         :  determina si el miembro tope de la pila contiene 0 (cero)
 * observaciones  :  - el código de esta función se utiliza en otras funciones
 */
int nPilaVacia(const t_pila *p)
{
   return p->tope == 0;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* return PILAVACIA(p);                                                    */
}


/*************************************************************************
 * función        :  nPonerEnPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 *                   t_info *d    - dirección de memoria en que se encuentra
 *                                  la información a poner en la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple (pila llena)
 * acción         :  por razones de seguridad, determina si no se ha utilizado
 *                      todo el espacio de almacenamiento destinado a la pila
 *                      antes de agregar la nueva información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
int nPonerEnPila(t_pila *p, const t_info *d)
{
   if(p->tope == MAX_PILA)
/* o mejor, en lugar del código anterior, utilizar el macro                */
/* if(PILALLENA(p))                                                        */
      return 0;
   else
   {
      p->pila[p->tope++] = *d;
      return 1;
   }
}


/*************************************************************************
 * función        :  nSacarDePila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información del tope de la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la pila está vacía
 *                      antes de recuperar la información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
```

Archivo : pila.c – (implementación estática)

```
int nSacarDePila(t_pila *p, t_info *d)
{
   if(p->tope == 0)
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* if(PILAVACIA(p))                                                        */
      return 0;
   else
   {
      *d = p->pila[--p->tope];
      return 1;
   }
}


/*****************************************************************************
 * función        :  nVerTopePila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información del tope de la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si no se ha utilizado
 *                      todo el espacio de almacenamiento destinado a la pila
 *                      antes de agregar la nueva información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
int nVerTopePila(t_pila *p, t_info *d)
{
   if(p->tope == 0)
/* o mejor, en lugar del código anterior, utilizar el macro                */
/* if(PILAVACIA(p))                                                        */
      return 0;
   else
   {
      *d = p->pila[p->tope - 1];
      return 1;
   }
}


/*****************************************************************************
 * función        :  vVaciarPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  n/a
 * acción         :  inicializa el miembro tope con 0
 * observaciones  :  - esta función coincide con la de Crear la Pila y su
 *                      código se repite en otras primitivas
 */
void vVaciarPila(t_pila *p)
{
   p->tope = 0;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* VACIARPILA(p);                                                          */
}
```

Archivo : cola.c – (implementación estática)

```
#include "cola.h"



#define CREARCOLA(X) ( (X)->pri =   0, (X)->ult   =  -1 )
```

Archivo : cola.c – (implementación estática)

```
#define VACIARCOLA   CREARCOLA


#define COLALLENA(X) ( (X)->pri ==  0 && (X)->ult == MAX_COLA - 1 || \
                       (X)->ult >  -1 && (X)->ult == (X)->pri - 1 )

#define COLAVACIA(X) ( (X)->ult == -1 )


/*****************************************************************************
 * función        :  vCrearCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  n/a
 * acción         :  inicializa el primero para la posicion 0
 *                      y al ultimo cn el indicador -1 (cola vacia)
 * observaciones  :  - esta función coincide con la de Vaciar la Cola y su
 *                      código se repite en otras primitivas
 */
void vCrearCola(t_cola *p)
{
   p->pri =  0;
   p->ult = -1;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* CREARCOLA(p);                                                           */
}


/*****************************************************************************
 * función        :  nColaLlena
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  1 (verdad) si está llena
 *                   0 (falso)  si no lo está
 * acción         :  determina si se han utilizado todas las posiciones del
 *                      array cola
 * observaciones  :  - el código de esta función se utiliza en otras funciones
 */
int nColaLlena(const t_cola *p)
{
   return p->pri ==  0 && p->ult == MAX_COLA - 1  ||
          p->ult >  -1 && p->ult == p->pri - 1;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* return COLALLENA(p);                                                    */
}


/*****************************************************************************
 * función        :  nColaVacia
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  1 (verdad) si está vacía
 *                   0 (falso)  si no lo está
 * acción         :  determina si no se ha utilizado ninguna de las posiciones
 *                       del array cola
 * observaciones  :  - el código de esta función se utiliza en otras funciones
 */
int nColaVacia(const t_cola *p)
{
   return p->ult == -1;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* return COLAVACIA(p);                                                    */
}
```

Archivo : cola.c – (implementación estática)

```
/*****************************************************************************
 * función        :  nPonerEnCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se encuentra
 *                                  la información a poner en la cola
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple (cola llena)
 * acción         :  por razones de seguridad, determina si no se ha utilizado
 *                      todo el espacio de almacenamiento destinado a la cola
 *                      antes de agregar la nueva información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
int nPonerEnCola(t_cola *p, const t_info *d)
{
   if(p->pri ==  0 && p->ult == MAX_COLA - 1  ||
      p->ult >  -1 && p->ult == p->pri - 1)
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* if(COLALLENA(p))                                                        */
      return 0;
   else
   {
      p->ult = (p->ult + 1) % MAX_COLA;
      p->cola[p->ult] = *d;
      return 1;
   }
}


/*****************************************************************************
 * función        :  nSacarDeCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información a sacar de la cola
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la cola está vacía
 *                      antes de recuperar la información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
int nSacarDeCola(t_cola *p, t_info *d)
{
   if(p->ult == -1)
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* if(COLAVACIA(p))                                                        */
      return 0;
   else
   {
      *d = p->cola[p->pri];
      if(p->pri != p->ult)
         p->pri = (p->pri + 1) % MAX_COLA;
      else
      {
         p->pri = 0;
         p->ult = -1;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/*       VACIARCOLA(p);                                                    */
      }
      return 1;
   }
}
```

Archivo : cola.c – (implementación estática)

```
/*****************************************************************************
 * función        :  nVerPrimeroCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información del primero de la cola, sin
 *                                  eliminarl de la misma
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la cola está vacía
 *                      antes de recuperar la información
 * observaciones  :  - parte del código de esta función se utiliza en otras
 *                      primitivas
 */
int nVerPrimeroCola(t_cola *p, t_info *d)
{
   if(p->ult == -1)
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* if(COLAVACIA(p))                                                        */
      return 0;
   else
   {
      *d = p->cola[p->pri];
      return 1;
   }
}


/*****************************************************************************
 * función        :  vVaciarCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  n/a
 * acción         :  inicializa el primero para la posicion 0
 *                      y al ultimo cn el indicador -1 (cola vacia)
 * observaciones  :  - esta función coincide con la de Crear la Cola y su
 *                      código se repite en otras primitivas
 */
void vVaciarCola(t_cola *p)
{
   p->pri =  0;
   p->ult = -1;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
/* VACIARCOLA(p);                                                          */
}
```

En este documento se describen dos archivos de 'cabecera' (**pila.h** y **cola.h**) y sus correspondientes desarrollos (**pila.h** y **cola.h**), para las implementaciones dinámicas de ambas estructuras de datos .

Archivo : pila.h – (implememtación dinámica)

```
#ifndef PILA_H__
#define PILA_H__


#include "info.h"


/*****************************************************************************
 * declaración del tipo de dato para los nodos de la pila compuesto de :
 *    información y un puntero al siguiente nodo en la pila (o mejor dicho, al
 *    cargado anteriormente)
 */
typedef struct s_nodo_pila
{
   t_info               info;
   struct s_nodo_pila  *sig;
} t_nodo_pila;


/*****************************************************************************
 * declaración del tipo de dato para manejar pilas dinámicas, el nuevo tipo de
 *    dato (t_pila) es en realidad un puntero a nodo, el que permitirá apuntar
 *    (es decir, tendrá la dirección de) al nodo que se encuentre en el tope
 *    de la pila (o NULL, si está vacía)
 */
typedef t_nodo_pila *t_pila;


/*****************************************************************************
 * primitivas para el manejo de pilas
 */
void vCrearPila(t_pila *p);

void vVaciarPila(t_pila *p);

int nPilaLlena(const t_pila *p);

int nPilaVacia(const t_pila *p);

int nPonerEnPila(t_pila *p, const t_info *d);

int nSacarDePila(t_pila *p, t_info *d);

int nVerTopePila(const t_pila *p, t_info *d);



#endif
```

Archivo : cola.h – (implememtación dinámica)

```
#ifndef COLA_H__
#define COLA_H__


#include "info.h"


/*****************************************************************************
 * declaración del tipo de dato para los nodos de la cola compuesto de :
 *    información y un puntero al siguiente nodo en la cola
```

Archivo : cola.h – (implememtación dinámica)

```
 */
typedef struct s_nodo_cola
{
   t_info               info;
   struct s_nodo_cola  *sig;
} t_nodo_cola;


/*****************************************************************************
 * declaración del tipo de dato para manejar colas dinámicas, el nuevo tipo de
 *    dato (t_cola) es en realidad una variable que contiene dos puntero a
 *    nodo, uno (pri) con la dirección del primero de la cola y el otro (ult)
 *    con la dirección del último
 */
typedef struct
{
   t_nodo_cola   *pri,
                 *ult;
} t_cola;

/*****************************************************************************
 * primitivas para el manejo de colas
 */
void vCrearCola(t_cola *p);

void vVaciarCola(t_cola *p);

int nColaLlena(const t_cola *p);

int nColaVacia(const t_cola *p);

int nPonerEnCola(t_cola *p, const t_info *d);

int nSacarDeCola(t_cola *p, t_info *d);

int nVerPrimeroCola(const t_cola *p, t_info *d);



#endif
```

Archivo : pila.c – (implememtación dinámica)

```
#include <stdlib.h>

#include "pila.h"




/*****************************************************************************
 * función        :  vCrearPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  n/a
 * acción         :  inicializa la pila con el valor NULL
 * observaciones  :
 */
void vCrearPila(t_pila *p)
{
   *p = NULL;
}


/*****************************************************************************
 * función        :  nPilaLlena
```

Archivo : pila.c – (implememtación dinámica)

```
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                   la pila (sólo por compatibilidad con la
 *                                   primitiva de asignación estática)
 * devuelve       :  1 (verdad)  si está llena (no hay más memoria dinámica)
 *                   0 (falso)   si no lo está (si aún hay memoria dinámica)
 * acción         :  obtiene memoria dinámica para un nodo y la libera, con
 *                      lo que puede determinar si había o no memoria
 * observaciones  :  - entre que se determina si hay o no memoria dinámica,
 *                   hasta que efectivamente se la use, otro proceso la pudo
 *                   consumir
 */
int nPilaLlena(const t_pila *p)
{
   void *nue = malloc(sizeof(t_nodo_pila));
   free(nue);
   return nue == NULL;
}


/*****************************************************************************
 * función        :  nPilaVacia
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                   la pila
 * devuelve       :  1 (verdad)  si está vacía
 *                   0 (falso)   si no lo está
 * acción         :  determina si la variable pila contiene o no NULL
 * observaciones  :  - el código de esta función se utiliza en otras funciones
 */
int nPilaVacia(const t_pila *p)
{
   return *p == NULL;
}


/*****************************************************************************
 * función        :  nPonerEnPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                   la pila
 *                   t_info *d    - dirección de memoria en que se encuentra
 *                                   la información a poner en la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple (falta memoria)
 * acción         :  por razones de seguridad, determina si no se ha utilizado
 *                      toda la memoria antes de agregar la nueva información
 * observaciones  :
 */
int nPonerEnPila(t_pila *p, const t_info *d)
{
   t_nodo_pila   *nue = (t_nodo_pila *)malloc(sizeof(t_nodo_pila));
   if(nue == NULL)
      return 0;
   else
   {
      nue->info = *d;
      nue->sig  = *p;
      *p        = nue;
      return 1;
   }
}


/*****************************************************************************
 * función        :  nSacarDePila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                   la pila
 *                   t_info *d    - dirección de memoria en que se almacenará
```

Archivo : pila.c – (implememtación dinámica)

```
 *                                  la información del tope de la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la pila está vacía
 *                      antes de recuperar la información
 * observaciones  :
 */
int nSacarDePila(t_pila *p, t_info *d)
{
   t_nodo_pila *elim;
   if(*p == NULL)
      return 0;
   else
   {
      elim = *p;
      *p = elim->sig;
      *d = elim->info;
      free(elim);
      return 1;
   }
}


/*****************************************************************************
 * función        :  nVerTopePila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información del tope de la pila
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la pila no está
 *                      vacía antes de recuperar la información del tope de la
 *                      pila
 * observaciones  :
 */
int nVerTopePila(const t_pila *p, t_info *d)
{
   if(*p == NULL)
      return 0;
   else
   {
      *d = (*p)->info;
      return 1;
   }
}


/*****************************************************************************
 * función        :  vVaciarPila
 * recibe         :  t_pila *p    - dirección de memoria en que se encuentra
 *                                  la pila
 * devuelve       :  n/a
 * acción         :  elimina (devolviendo al sistema operativo) la memoria
 *                      ocupada por todos los nodos dejando, al final, la pila
 *                      vacía (con NULL)
 * observaciones  :
 */
void vVaciarPila(t_pila *p)
{
   t_nodo_pila  *elim;
   while(*p)
   {
      elim = *p;
      *p   = elim->sig;
      free(elim);
```

Archivo : pila.c – (implememtación dinámica)

```
   }
}
```

Archivo : cola.c – (implememtación dinámica)

```
#include <stdlib.h>


#include "cola.h"


#define COLAVACIA(X) ( (X)->pri ==  NULL )
/* o de lo contrario . . .
#define COLAVACIA(X) ( (X)->ult ==  NULL )
 */


/*****************************************************************************
 * función        :  vCrearCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  n/a
 * acción         :  inicializa el primero y último con NULL (vacía)
 * observaciones  :  -
 */
void vCrearCola(t_cola *p)
{
   p->pri =  p->ult = NULL;
}


/*****************************************************************************
 * función        :  nColaLlena
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  1 (verdad) si está llena
 *                   0 (falso)  si no lo está
 * acción         :  determina si se ha utilizado toda la memoria
 * observaciones  :  - el argumento sólo se recibe por compatibilidad con la
 *                   primitiva de asignación estática de memoria y no se lo
 *                   utiliza
 */
int nColaLlena(const t_cola *p)
{
   void *nue = malloc(sizeof(t_nodo_cola));
/* si su compilador da un warning debido a que el argumento que se recibe  */
/* no se utiliza, podrá agregar la siguiente línea                          */
/* p = NULL;                                                                */
   free(nue);
   return nue == NULL;
}


/*****************************************************************************
 * función        :  nColaVacia
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  1 (verdad) si está vacía
 *                   0 (falso)  si no lo está
 * acción         :  determina si no hay primero (o último)
 * observaciones  :
 */
int nColaVacia(const t_cola *p)
{
   return p->pri == NULL;
/* o mejor utilizar el macro ('visible' sólo por este archivo)             */
```

Archivo : cola.c – (implememtación dinámica)

```
/* return COLAVACIA(p);                                                    */
}


/*************************************************************************
 * función        :  nPonerEnCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se encuentra
 *                                  la información a poner en la cola
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple (no hay memoria)
 * acción         :  por razones de seguridad, determina si no se ha utilizado
 *                      todo la memoria antes de agregar la nueva información
 * observaciones  :
 */
int nPonerEnCola(t_cola *p, const t_info *d)
{
   t_nodo_cola *nue = (t_nodo_cola *)malloc(sizeof(t_nodo_cola));

   if(nue == NULL)
      return 0;
   else
   {
      nue->info = *d;
      nue->sig = NULL;
      if(p->pri == NULL)
/* o mejor utilizar el macro ('visible' sólo por este archivo)            */
/*    if(COLAVACIA(p))                                                    */
         p->pri = nue;
      else
         p->ult->sig = nue;
      p->ult = nue;
      return 1;
   }
}


/*************************************************************************
 * función        :  nSacarDeCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información a sacar de la cola
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la cola está vacía
 *                      antes de recuperar la información
 * observaciones  :
 */
int nSacarDeCola(t_cola *p, t_info *d)
{
   t_nodo_cola *elim;

   if(p->pri == NULL)
/* o mejor utilizar el macro ('visible' sólo por este archivo)            */
/* if(COLAVACIA(p))                                                       */
      return 0;
   else
   {
      elim = p->pri;
      *d = elim->info;
      p->pri = elim->sig;
      free(elim);
      if(p->pri == NULL)
/* o mejor utilizar el macro ('visible' sólo por este archivo)            */
```

Archivo : cola.c – (implememtación dinámica)

```
/*    if(COLAVACIA(p))                                                    */
         p->ult = NULL;
      return 1;
   }
}


/*************************************************************************
 * función        :  nVerPrimeroCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola, y
 *                   t_info *d    - dirección de memoria en que se almacenará
 *                                  la información del primero de la cola, sin
 *                                  eliminarlo de la misma
 * devuelve       :  1 (verdad) si cumple su cometido
 *                   0 (falso)  si no lo cumple
 * acción         :  por razones de seguridad, determina si la cola está vacía
 *                      antes de recuperar la información
 * observaciones  :  -
 */
int nVerPrimeroCola(const t_cola *p, t_info *d)
{
   if(p->pri == NULL)
/* o mejor utilizar el macro ('visible' sólo por este archivo)            */
/* if(COLAVACIA(p))                                                       */
      return 0;
   else
   {
      *d = p->pri->info;
      return 1;
   }
}


/*************************************************************************
 * función        :  vVaciarCola
 * recibe         :  t_cola *p    - dirección de memoria en que se encuentra
 *                                  la cola
 * devuelve       :  n/a
 * acción         :  inicializa el primero para la posicion 0
 *                      y al ultimo cn el indicador -1 (cola vacia)
 * observaciones  :  -
 */
void vVaciarCola(t_cola *p)
{
   t_nodo_cola *elim;

   while(p->pri)
/* o mejor utilizar el macro ('visible' sólo por este archivo)            */
/* while(!COLAVACIA(p))                                                   */
   {
      elim = p->pri;
      p->pri = elim->sig;
      free(elim);
   }
   p->ult = NULL;
}
```