Trabajo Práctico
Tipo Abstracto de Dato Número Astronómico

Sintaxis y Semántica de los Lenguajes
K2003

Dr. [REDACTED] - Ing. [REDACTED]

Integrantes:

Dipietro, Guido - 000.000-0
Irigaray, Magalí - 000.000-0

Universidad Tecnológica Nacional

Facultad Regional Buenos Aires
octubre de 2020

Índice

1. Objetivos

- Presentar el concepto de biblioteca
- Presentar los pasos necesarios para la creación de una biblioteca con ANSI C
- Presentar los pasos para la construcción de biblioteca con BCC32
- Presentar los pasos para compilar (y linkeditar) con BCC32 programas fuente que utilizan bibliotecas aparte de la Standard

2. Introducción

En este trabajo práctico realizaremos una aplicación práctica del concepto de bibliotecas en C que consistirá en la implementación de un Tipo Abstracto de Dato (de ahora en más a ser referido como *TAD*) denominado *Número Astronómico*.

Este *TAD* permitirá representar números enteros muy grandes que no son posibles de manejar con el tipo de dato nativo de ANSI C más grande, el *unsigned long long*[1] de 8 bytes, con un valor máximo de $2^{64} - 1 = 18,446,744,073,709,551,615$ (número de 20 cifras).

Un Número Astronómico tendrá la capacidad de utilizar enteros positivos de hasta 100 cifras, permitiendo las operaciones de *suma*, *comparación por igual*, y *comparación por menor*.

Además, nuestra biblioteca permitirá la creación de Números Astronómicos de tres formas distintas:

- Desde cadena
- Desde cifra seguida de ceros
- Aleatorio

Se podrá salvar y recuperar los números astronómicos creados a archivos binarios y de texto, así como mostrarlos en pantalla en grupos.

3. Implementación en ANSI C

[2]Utilizaremos las siguientes bibliotecas para su implementación:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <time.h>
7  #include "astronum.h"
```

El archivo de cabecera `astronum.h` con los prototipos de nuestras funciones se exhibe completamente en la sección Apéndice, página 20.

[1]Disponible en C99

[2]Los comentarios en los programas no tendrán tildes en ninguna vocal, dado que esto causa errores a la hora de renderizar el documento en LaTeX.

3.1. Definición del TAD

El *TAD* será modelado como un *struct* con los siguientes campos:

```
typedef struct {
    char* entero; // contiene el entero
    int longitudError;  // contiene el largo de la cadena, o el error
} NumeroAstronomico;
```

Siendo los tipos de errores un tipo de dato enumerado como sigue:

```
typedef enum {
    //Ninguno,          //sin codigo, longitudError > 0
    CadenaInvalida=-3,  // -3
    Overflow,           // -2
    PunteroNulo,        // -1
    CadenaNula          //  0 (coincide con el largo en este caso)
} TipoDeError;
```

Elegimos los enteros equivalentes a cada opción del `enum` de una forma inteligente para que los tipos de errores fueran enteros negativos, y para que `CadenaNula` coincidiera con la longitud de la cadena (0).
En nuestra implementación, la `CadenaNula` no siempre será un error, sino que es posible realizar algunas operaciones con ella (por ejemplo, al sumar se toma como 0).

Si el campo `longitudError` es mayor a 0, éste representará la longitud del entero almacenado en `entero` y señalará que no hay ningún error.

3.2. Creación

Veremos las tres distintas funciones de creación que posee esta biblioteca:

- `CrearDesdeCadena(char*)`
- `CrearDesdeCifraSeguidaDeCeros(int, int)`
- `CrearAleatorio()`

3.2.1. Crear desde cadena

Veamos primero la función `CrearDesdeCadena(char*)` que, dado un puntero a `char`, creará el Número Astronómico (ya sea con o sin errores).

Al ser un requerimiento del Trabajo Práctico la utilización de `malloc` para reservar dinámicamente la memoria, todas nuestras funciones de creación retornarán un puntero a Número Astronómico, concretamente, un `NumeroAstronomico*`.

Para reservar la memoria, se intentará realizar el `malloc` para el puntero del `struct`. Si falla, se retornará un puntero nulo.

```
NumeroAstronomico* CrearDesdeCadena(char* cadena){
    // malloc del STRUCT
    NumeroAstronomico *num = malloc(sizeof(NumeroAstronomico));
    if (num==NULL) return NULL; // si falla
```

Ningún Número Astronómico puede comenzar con un char `'0'`; esto se contempla a continuación:

```
    for(int i=0; cadena[i]=='0'&&cadena[i]!='\0'; *(cadena)++);
```

Si la cadena brindada solamente está compuesta por ceros, el comportamiento es indefinido.[3].

Luego, se evalúan otros tipos de errores (o la creación exitosa) y se retorna un puntero a Número Astronómico con el estado que corresponda:

```
        // Caso cadena muy larga (>100 caracteres)
        if (strlen(cadena)>100) {
            num->entero = strdup("");
            num->longitudError = Overflow;
        }
        // Caso todo bien (o cadena nula)
        else if (cadenaNumerica(cadena)){
            char* clean_cadena = strdup(cadena);
            num->entero = clean_cadena;
            num->longitudError = strlen(clean_cadena);
        }
        // Caso cadena invalida
        else {
            num->entero = strdup("");
            num->longitudError = CadenaInvalida;
        }
        return num;
    }
```

En los casos de error, la cadena será inicializada a una cadena vacía, a modo de no dejar un puntero con basura.

La función `cadenaNumerica(char*)` es privada y luce así:

```
    // retorna (todos sus caracteres son numericos? 1 : 0)
    static int cadenaNumerica(char* cadena){
        for (int out=0; cadena[out]!='\0'; out++)
            if (!isdigit(cadena[out])) return 0;
        return 1;
    }
```

Las otras dos opciones de creación utilizan la función `CrearDesdeCadena(char*)`, para evitar repetir toda la lógica de reserva de memorias y el manejo de errores.

3.2.2. Crear desde entero seguido de ceros

Si bien implementamos esta función para que funcione con un entero (no necesariamente de una sola cifra) seguido de ceros, conservamos el nombre original que se encuentra en la consigna.

`CrearDesdeCifraSeguidaDeCeros(int, int)` creará una cadena en base a lo que se le indique[4], y luego retornará un `NumeroAstronomico*` creado con `CrearDesdeCadena(char*)` utilizando la cadena que generó.

[3]Este *TAD* fue originalmente definido solo para enteros positivos.

[4]Si la cifra indicada es un cero, se creará un Número Astronómico de valor nulo. Si bien originalmente el *TAD* se diseñó solo para enteros positivos, posteriores arreglos de *memory leaks* llevaron a tomar esa decisión.

Como precondición, fijamos que la cantidad de "ceros" pedida no puede superar los 99, y que la longitud total de la cadena no puede superar los 100.
Ante esas situaciones, retornará `NULL` inmediatamente.

```
1  NumeroAstronomico* CrearDesdeCifraSeguidaDeCeros(int num, int ceros){
2      // Evaluacion de precondiciones
3      if (ceros>99) return NULL;
4      if (num==0) return CrearDesdeCadena("0");
5      int longitud_num = (int) log10(num) + 1;    // cantidad de digitos de
   num
6      int longitud_total =  longitud_num + ceros;
7      if (longitud_total > 100) return NULL;
8
9      // Creacion de la cadena
10     char* cadena = malloc(longitud_total+1);
11     if(cadena==NULL) return NULL; // fallo malloc
12     sprintf(cadena, "%d", num); // copia num
13
14     for(int i=0; i<ceros; i++) cadena[i+longitud_num]='0'; // llena el resto
    con 0
15
16     cadena[longitud_total] = '\0'; // fin de cadena
17
18     NumeroAstronomico* temp = CrearDesdeCadena(cadena); // para evitar
   repetir el codigo del malloc
19     free(cadena);
20     return temp;
21 }
```

3.2.3. Crear aleatorio

Finalmente, `CrearAleatorio()` hará uso de las funciones `rand()` y `srand(unsigned int)` para generar una cadena aleatoria de aleatoria entre 1 y 100 [5].

Luego, retornará el `NumeroAstronomico*` usando `CrearDesdeCadena(char*)`.

Usaremos el tiempo local que retorne `time(time_t*)` como *seeding* para la función `rand()`.

```
1  NumeroAstronomico* CrearAleatorio(){
2      srand(time(NULL)); // seeding para mayor aleatoriedad
3
4      int longitud = 1 + rand() %100; // entre 1 y 100
5      char* cadena = malloc(longitud+1);
6      if(cadena==NULL) return NULL; // fallo malloc
7
8      // Creacion de cadena
9      for(int i=0; i<longitud; i++){
10         char cifra_aleatoria = nac(rand()%10);
11         cadena[i] = cifra_aleatoria;
12     }
13     cadena[longitud] = '\0';
14     NumeroAstronomico* temp = CrearDesdeCadena(cadena);
15     free(cadena); return temp;
16 }
```

[5] Esta función solamente genera enteros sin error (por eso no se permite el Número nulo). Además, se precisa un byte más en memoria para el caracter de fin de cadena.

Algo a comentar es que, como evidentemente el *seeding* es equivalente en ejecuciones que sucedan dentro del mismo segundo, se obtendrán los mismos resultados en programas que se ejecuten dentro del mismo segundo (por ejemplo, a las 14:27:49.07 y a las 14:27:49.73).

Otra cosa a notar es que `nac(x)` es una *Macro* que convierte un dígito entero a su versión en carácter. También definimos la operación inversa:

```
// Macros caracter a int y viceversa
#define can(c) (c-'0')
#define nac(n) (n+'0')
```

Utilizaremos estas *Macros* de nuevo más adelante.

3.2.4. Liberar memoria

En todos los casos, la memoria reservada dinámicamente para el `NumeroAstronomico*` puede ser liberada mediante la siguiente función:

```
void* FreeNumeroAstronomico(NumeroAstronomico* a){
    free(a->entero);
    free(a);
}
```

3.3. Aritmética

Detallaremos la implementación de las tres funciones de aritmética de nuestra biblioteca:

- `SonIguales`
- `EsMenor`
- `Sumar`

3.3.1. Comparaciones

Decidimos que la comparación por igual entre Números Astronómicos retornará 1 en caso de que:

- Sean ambos válidos y tengan el mismo entero, o bien
- Tengan ambos el mismo tipo de error

Se implementa así:

```
int SonIguales(NumeroAstronomico* a, NumeroAstronomico* b){
    if(EsError(a) && EsError(b))
        return a->longitudError == b->longitudError;

    if(a->longitudError != b->longitudError) return 0;
    int largo = a->longitudError; // si llegan aca, son iguales en largo
    int i=0;

    for (i; ((i<largo) && (a->entero[i]==b->entero[i])); i++);
    return i==largo;
}
```

Las funciones de control de errores (como `EsError(NumeroAstronomico*)`) están disponibles en la sección Apéndice, página 21.

La comparación por menor retorna 1 si el primer Número Astronómico dado es menor al segundo, y 0 en otro caso (también retorna 0 si alguno de los dos tiene un error).

Su código:

```
1   int EsMenor(NumeroAstronomico* a, NumeroAstronomico* b){
2       if(EsError(a) || EsError(b)) return 0;
3
4       if (a->longitudError < b->longitudError)
5           return 1; //"a" es mas corto
6       else if (a->longitudError > b->longitudError)
7           return 0; //"a" es mas largo
8       else {
9           // son iguales en longitud
10          for(int i=0; i<a->longitudError; i++)
11              if (a->entero[i] < b->entero[i]) return 1;
12              //"a" misma longitud que "b" pero menor
13      }
14
15      return 0; //Si llega hasta aca, son iguales (entonces no es menor)
16  }
```

Es posible comparar con < dos `char` (línea 11) ya que `char` es un subconjunto de `int`, y además, los carácteres que representan a los dígitos del 0 al 9 están en la tabla ASCII ordenados de forma creciente.

La función `longmax(NumeroAstronomico* a, NumeroAstronomico* b)` es propia de la implementación y muy sencilla; solamente la escribimos para evitar tipear tanto texto:

```
1   static int longmax(NumeroAstronomico* a, NumeroAstronomico* b){
2       return (a->longitudError > b->longitudError? a->longitudError : b->
    longitudError);
3   }
```

3.3.2. Suma

Finalmente, la función `Sumar(NumeroAstronomico*, NumeroAstronomico*)` retornará el puntero a Número Astronómico creado por `CrearDesdeCadena(char*)`, pasándole como argumento la "suma"de ambas cadenas (realizado con una función propia que explicaremos posteriormente).

En caso de que dicha cadena tenga una longitud mayor a 100, `CrearDesdeCadena(char*)` se encargará de retornar un `NumeroAstronomico*` con el error `Overflow`.

Por otro lado, en caso de que uno o ambos Números Astronómicos tengan la cadena nula, se sumarán como si fuera el neutro de la suma (el número 0).

Si ambos fueran nulos, lo que se retornará será también un Número Astronómico nulo.

En estos tres casos, se retornará un nuevo Número Astronómico para evitar apuntar a la misma dirección en memoria que el número que se pasa.[6]

[6]Cabe destacar que esto dio lugar a un problema en el uso práctico de la biblioteca, que cubriremos más adelante.

Además, si alguno de los números tiene otro tipo de error, se informará en pantalla y retornará un puntero nulo.

Su código:

```
NumeroAstronomico* Sumar(NumeroAstronomico* a, NumeroAstronomico* b){
    if(a==NULL || b==NULL) return NULL;
    if (EsSecuenciaInvalida(a) || EsSecuenciaInvalida(b)){
        fprintf(stderr, "Alguno de los numeros sumados es invalido.\n");
        NumeroAstronomico* invalido = malloc(sizeof(NumeroAstronomico));
        invalido->entero = strdup("");
        invalido->longitudError = CadenaInvalida;
        return invalido;
    }
    if (EsSecuenciaNula(a) && !EsError(b))
        return CrearDesdeCadena(b->entero); // se toma como sumar 0
    if (EsSecuenciaNula(b) && !EsError(a))
        return CrearDesdeCadena(a->entero); // lo mismo
    if (EsSecuenciaNula(a) && EsSecuenciaNula(b))
        return CrearDesdeCadena(""); // si los dos son nulos retorna nulo

    if (!EsError(a) && !EsError(b)){ // todo en orden
        char* suma = sumaCadenas(a->entero, a->longitudError, b->entero, b->
longitudError);
        // Si hubo Overflow (>100 caracteres), CrearDesdeCadena lo va a
manejar
        NumeroAstronomico* temp = CrearDesdeCadena(suma);
        free(suma); return temp;
    }
    else { // algun error no manejado
        fprintf(stderr, "Ocurrio un error en la suma.\n");
        return NULL;
    }
}
```

La función privada `sumaCadenas(char*, char*)` retorna un puntero a `char` con la suma de las dos cadenas dadas, como si fueran enteros.

El método de suma que usaremos será cifra por cifra, de izquierda a derecha, arreglando los acarreos después de sumar cada cifra.

Comenzaremos reservando memoria para una cadena que tenga longitud[7] igual a $2 + longmax$, siendo $longmax$ la longitud máxima de las cadenas de los enteros de ambos Números Astronómicos.

Al reservar una cadena de esa longitud, estamos contemplando el caso máximo (por ejemplo $99 + 1 = 100$, un número de 2 cifras sumado a otro de 1 dan uno de 3. No podrían dar nunca uno de 4).

```
static char* sumaCadenas(const char* a, int lena, const char* b, int lenb){
    // Longitud maxima, cadena larga, cadena corta
    int longmax = lena>lenb? lena : lenb;
    longmax++;
    char* corta = lena<lenb? strdup(a) : strdup(b);
    char* larga = lena<lenb? strdup(b) : strdup(a);
    char* salida = malloc(longmax+1);
```

[7] Un 1 extra para el carácter de fin de cadena, y 1 extra para contemplar la longitud máxima de una suma de dos enteros.

Para poder sumar cifra a cifra, necesitamos alinear ambas cadenas a derecha. Para conseguir eso, se rellena la cadena más corta con ceros a la izquierda.

Sumaremos comenzando en el segundo carácter[8], a modo de dejar que la cadena comience en 0 en caso de precisar la longitud máxima.

Por cada cifra que sume, además, se arreglarán los *carries* siguiendo el siguiente algoritmo:

1. Si la suma de las dos cifras es mayor a 9, tomar solamente su unidad y sumarle 1 a la unidad anterior
2. Repetir este algoritmo con la unidad anterior mientras estas suman resulten mayores a 9

Además, por supuesto hay que adicionar el carácter de fin de cadena al final de esto.

En caso de haber "sobreestimado" la longitud máxima (por ejemplo, $14 + 27 = 41$), la cadena final comenzará en 0.
Para evitar esto, aumentaremos el valor del puntero en 1 para que apunte a la primera cifra significativa (cambiando, por ejemplo, un retorno de "041" a "41").

Un ejemplo de este procedimiento:

1. 1427 + 4927
2. Salida: 00000?
3. Primera suma (1+4): 05000?
4. Segunda suma (4+9 con carry): 05300? → 06300?
5. Tercera suma (2+2): 06340?
6. Cuarta suma (7+7 con carry): 06344? → 06354?
7. Fin de cadena: 06354\0
8. Arreglar puntero: 6354\0 (posterior magia malloc para evitar el error `free(): invalid pointer`)

Su implementación en C (con las Macros que definimos en la subsección 3.2 Creación pág. 6):

```
    // Diferencia entre la larga y la corta (para rellenar con 0s)
    int diff = longmax - strlen(corta);

    // A sumar!
    int buffer;                     // auxiliar para carries
    int carries;                    // auxiliar para carries parte 2
    salida[0] = '0';                // primera cifra en 0
    // Suma caracter a caracter (de izquierda a derecha)
    for(int i=1; i<longmax; i++){
        char padding_or_num = i<diff? '0' : corta[i-diff];
        // 0 o digito (cadena corta)
        int cifra = can(padding_or_num) + can(larga[i-1]);
        // digito (cadena larga)

        // asigna el valor char de esa cifra (solo las unidades)
        salida[i] = nac(cifra%10);

```

[8] ¿Por qué no sumar empezando por el último? El Guido del futuro se pregunta lo mismo.

```
18          // arregla los carries (suma 1 para atras con cada carry que surja)
19          buffer = cifra; // variable auxiliar para un poco de claridad
20          carries = 1;
21          while(buffer > 9){
22              buffer = can(salida[i-carries]+1);
23              salida[i-carries] = nac(buffer%10);
24              carries++;
25          }
26      }
27      salida[longmax] = '\0'; // Fin de cadena
28
29      char* out = salida[0]=='0'? strdup(salida+1) : strdup(salida);
30      // Longitud sobreestimada
31
32      free(salida); free(corta); free(larga);
33      return out;
34  }
```

El lector sabrá disculpar la aberración, inmundicia, desperdicio, porquería, bazofia, despojo de código que acaba de leer.
Quien lo implementó, que es la misma persona que escribe este párrafo, no tiene idea de qué tenía en la cabeza al momento de hacerlo. Este algoritmo es horrible.

Por favor, hágase un favor y olvídelo para siempre.

3.4. Salida

Hay una única operación de salida denominada `Mostrar(NumeroAstronomico*, int)`, que recibe un puntero a Número Astronómico, y la cantidad de grupos (de a 3 cifras) que debe mostrar en la primera línea.

Algunos ejemplos son:

```
1  NumeroAstronomico* a = CrearDesdeCifraSeguidaDeCeros(618, 26);
2  NumeroAstronomico* chiquito = CrearDesdeCadena("7373");
3
4  Mostrar(a, 5);
5  /*
6   61.800.000.000.000.
7      000.000.000.
8      000.000.
9  */
10
11 Mostrar(a, 3);
12 /*
13  61.800.000.
14     000.000.000.
15     000.000.000.
16     000.
17 */
18
19 Mostrar(chiquito, 69);
20 // no tiene tantos grupos pero no da error
21 /*
22  7.373.
23 */
```

En su implementación, si es algún tipo de error imprime en `stderr` que se trata de ello:

```
void Mostrar(NumeroAstronomico* a, int grupos){
    // Si hay algun error en el numero
    // es mas facil detectarlo asi que usando las funciones EsTipoError()...
    if(a == NULL) { fprintf(stderr, "Puntero nulo\n"); return; }
    switch(a->longitudError){
        case Overflow: fprintf(stderr, "Overflow\n"); return;
        case PunteroNulo: fprintf(stderr, "Puntero nulo\n"); return;
        case CadenaInvalida: fprintf(stderr, "Cadena invalida\n"); return;
        case CadenaNula: fprintf(stderr, "Cadena nula\n"); return;
    }
```

Mostraremos la primera línea utilizando `putchar(char)`, por lo que en primer lugar se necesita calcular cuántos espacios imprimir para que la primera línea coincida con las subsiguientes (en cuanto a indentación).

El número de grupos a mostrar en las líneas después de la primera se define mediante un `#define`:

```
#define MOSTRAR_GRUPOS_LINEAS 3
```

El cálculo de estos parámetros y *printeo* de los valores que impliquen es como sigue:

```
    // Primer grupo, primera linea
    int espacios = (3-(a->longitudError %3)) %3;
    int i = 0-espacios;
    for (i; i<3-espacios; i++)
        (i<0)? putchar(' ') : putchar(a->entero[i]);
    putchar('.');
    // Terminar de mostrar primera linea
    mostrarNGrupos(a, grupos-1, &i);
    // Muestra el resto
    // (indentado para que quede debajo del 2do grupo de la primera linea)
    while(i < a->longitudError){
        printf("    ");
        mostrarNGrupos(a, MOSTRAR_GRUPOS_LINEAS, &i);
    }
    putchar('\n');
}
```

Dos cosas a notar:

La primera, es evidente que en la matemática $(3 - (x \mod 3)) \mod 3 = (-x) \mod 3$, pero en ANSI C esto no se cumple. Un ejemplo de ello:

```
#include <stdio.h>
int main(){
    int a = 16;
    int una = (3-(a%3))%3;
    int otra = (-1*a)%3;
    printf("Una: %d, Otra: %d\n", una, otra);
    // Una: 2, Otra: -1
    return 0;
}
```

De ahí la expresión extraña para el cálculo de los espacios.

La segunda, es que la función `mostrarNGrupos(NumeroAstronomico*, int, int*)`, que muestra N grupos de una cadena a partir de un índice dado, se define como sigue:

```c
static void mostrarNGrupos(NumeroAstronomico* a, int cant, int *ind){
    int tope = (*ind) + cant*3; //hasta que indice mostrar
    int comienzo = (*ind); //en cual se empezo originalmente

    for (*ind; (((*ind) < tope) && (a->entero[*ind] != '\0')); (*ind)++){
        // este bucle cuenta la cantidad de digitos segun 'cant'
        // y rompe si termino la cadena (!='\0')
        putchar(a->entero[(*ind)]);
        if ((*ind - comienzo)%3 == 2) putchar('.');
    }
    putchar('\n');
}
```

El índice se pasa como un puntero por el uso que se le da en la función anterior (es necesario que el índice aumente para poder llamar a la función `mostrarNGrupos`) reiteradas veces en el bucle `while`).

3.5. Persistencia

Existen funciones para escribir en modo binario y texto, que se corresponden a lo siguiente:

$$\begin{aligned} \texttt{Scan} &\iff \texttt{fscanf} \\ \texttt{Print} &\iff \texttt{fprintf} \\ \texttt{Read} &\iff \texttt{fread} \\ \texttt{Write} &\iff \texttt{fwrite} \end{aligned}$$

3.5.1. Texto (`Scan` y `Print`)

El formato que le daremos a los Números Astronómicos en texto es el siguiente:

$$\begin{cases} \text{Si no es error:} & \texttt{<entero> <longitudError>} \\ \text{Si es error:} & \texttt{error <longitudError>} \end{cases}$$

Para conseguir eso, la función se implementa como sigue:

```c
void Print(FILE* f, NumeroAstronomico* a){
    if(a==NULL || f==NULL) return;
    if(a->longitudError <= 0) //si tiene error
        fprintf(f, "error %d\n", a->longitudError);
    else
        fprintf(f, "%s %d\n", a->entero, a->longitudError);
}
```

La palabra `error` solamente se incluye para que la función `Scan` pueda leer con formato sin problemas.

`Scan` leerá el entero y la cadena a la vez, por lo que creamos un *buffer* con el tamaño máximo (100 caracteres):[9]

[9] No es necesario hacer un `free` porque se almacena en la pila.

```c
int Scan(FILE* f, NumeroAstronomico* a){
    if(a==NULL || f==NULL) return 0;
    // buffers
    char cadena[100];
    int longitud = 0;
    // lee longitud
    int leido = fscanf(f, "%s %d\n", cadena, &longitud);
    if(longitud > 0) {//si no tiene error
        free(a->entero);
        a->entero = strdup(cadena);
    }
    a->longitudError = longitud;

    return leido; // para conocer EOF
}
```

Esta función retorna un `int` con la cantidad de elementos leídos, o -1 si terminó, para poder aplicarla por ejemplo así:

```c
#include <stdlib.h>
#include "astronum.h"

int main(){
    FILE* f = fopen("file.txt", "r");
    NumeroAstronomico* temp = malloc(sizeof(NumeroAstronomico));

    while(Scan(f,temp) != -1){
        Mostrar(temp, 3);
    }
    FreeNumeroAstronomico(temp);

    return 0;
}
```

Vemos que es posible escribir y leer Números Astronómicos tanto válidos como con error, sin ningún tipo de problemas.

3.5.2. Binario (`Read` y `Write`)

En este caso, podemos hacer funciones un poco más eficientes que las anteriores.

Para escribir a un archivo, primero escribiremos el campo `longitudError`, y luego el entero **si y solo si** no tiene error. Porque, si tiene error, ¿cuál es el sentido de escribir la cadena?

Su implementación:

```c
void Write(FILE* f, NumeroAstronomico* a){
    fwrite(&(a->longitudError), sizeof(int), 1, f);

    if (a->longitudError > 0)
        fwrite(a->entero, sizeof(char), 1+(a->longitudError), f);
}
```

Cabe resaltar que estamos escribiendo además el carácter de fin de cadena `\0`, he allí la razón por la cual escribimos "*1 más*" que la longitud de cadena.

Haber hecho esto así permite implementar la función de lectura binaria `Read` como sigue:

```
size_t Read(FILE* f, NumeroAstronomico* a){
    // Primero leo longitud / error
    int longitud = 0;
    size_t bytes_longitud = fread(&longitud, sizeof(int), 1, f);
    size_t bytes_cadena = 0;

    // Si no es error
    if (longitud > 0){
        char* cadena = malloc(longitud+1);
        bytes_cadena = fread(cadena, sizeof(char), 1+longitud, f);
        free(a->entero);
        a->entero = cadena;
    }
    // Si es error, solo necesito el codigo de error
    a->longitudError = longitud;

    return bytes_cadena + bytes_longitud; //para conocer EOF
}
```

En este caso, a diferencia del `Scan`, `Read` retornará un tipo `size_t` con la cantidad de *bytes* leídos. Cuando no haya más que leer, retornará 0, por lo que su uso es un poco más simple que la función de texto:

```
#include <stdlib.h>
#include "astronum.h"

int main(){
    FILE* f = fopen("file.txt", "rb");
    NumeroAstronomico* temp = malloc(sizeof(NumeroAstronomico));

    while(Read(f,temp)){
        Mostrar(temp, 3);
    }
    FreeNumeroAstronomico(temp);

    return 0;
}
```

El modo binario también permite la lectura y escritura de Números Astronómicos tanto válidos como con error, sin ningún problema.

Con esto concluímos el detalle de la implementación de todas las funciones en la biblioteca Número Astronómico.

Ver la sección Apéndice, página 15 para más ejemplos de la implementación, o en la página 15 para seguir el enlace al repositorio en GitHub con el código completo de todo el Trabajo Práctico.

4. Epílogo

Y así fue como el trabajo se entregó, no fue nunca corregido, y la materia de la que surgió su enunciado fue aprobada sin mayores dificultades.

Al terminar el año de la forma más amena posible, pandemia mediante, y transitar el verano dejando fluir los problemas del 2020, tuve la idea de revisar este Trabajo para ver qué había sido de él.

Tamaña sorpresa recibí al ejecutar el comando

```
valgrind ./principal
```

y ver que la cantidad de *memory leaks* hacía que mi programa pareciera un colador de memoria.

Eso me llevó a reflexionar. Es decir, a experimentar la reflexión de la luz al mirarme al espejo para ver la cara de quien había programado semejante desastre. Pista: fui yo.

Pasé un rato solucionando los problemas que existían, y lamentablemente esto estropeó la calidad del código.
Si bien los *memleaks* se fueron para nunca más volver, el estilo de código se vio corrompido, y muchas implementaciones o *tests* realizados en el programa principal involucraron una fea cantidad de código repetido, o desprolijo sin más.

¿Es eso preferible a un programa más organizado pero lleno de *bugs* invisibles? Posiblemente.

¿Es la estrategia óptima? Quizás lo es para el contexto dado.[10]

¿Cuál es la estrategia óptima, independientemente del contexto? Refactorizar todo el código. Escribir las funciones problemáticas de nuevo, desde cero. Repensar el funcionamiento y alcance de la biblioteca y el *TAD*.

¿Se hará eso alguna vez? No.

Gracias por su lectura, y espero que este informe haya sido de su agrado.

5. Apéndice

5.1. Repositorio en GitHub

En este repositorio se encuentra todo el código referido a este Trabajo Práctico.

```
https://github.com/GuidoDipietro/TAD_NumeroAstronomico
```

5.2. Ejemplos de uso

Este programa puede ser ejecutado para obtener el resultado que se comenta al lado de cada línea.

Compilar utilizando el siguiente comando para generar el ejecutable `main`.

```
gcc -o main principal.c astronum.c -g -I.
```

[10] Contexto: no tengo ganas de invertir tanto tiempo en un TP de un año anterior. No tiene sentido.

Para ver el código fuente, o en un formato un poco más prolijo, dirijirse al siguiente archivo en el repositorio en GitHub:

`https://github.com/GuidoDipietro/TAD_NumeroAstronomico/blob/master/principal.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "astronum.h"

int main(){
    clock_t begin = clock(); //tomemos el tiempo que tarda

    NumeroAstronomico* p1 = CrearDesdeCadena("12345678900000000000000000"); // 17 ceros
    NumeroAstronomico* p1_con_ceros = CrearDesdeCadena("00001234567890000000000000000000"); // 17 ceros
    NumeroAstronomico* p2 = CrearDesdeCadena("123456789000000000000000000"); // 18 ceros
    NumeroAstronomico* p3 = CrearDesdeCadena("1234567890000000000000000000"); // 19 ceros
    NumeroAstronomico* p3_equivalente = CrearDesdeCifraSeguidaDeCeros(123456789, 19);
    NumeroAstronomico* cosaNula = CrearDesdeCadena("");
    NumeroAstronomico* invalido = CrearDesdeCadena("buenas123");
    NumeroAstronomico* otro_invalido = CrearDesdeCadena("aguantePython");
    NumeroAstronomico* chiquito = CrearDesdeCadena("1427");
    NumeroAstronomico* pequenio = CrearDesdeCadena("1427");
    NumeroAstronomico* otropeque = CrearDesdeCadena("4969");
    NumeroAstronomico* largo1 = CrearDesdeCadena("11111111111111111111"); //20 "1"s
    NumeroAstronomico* largo2 = CrearDesdeCadena("11111111111111111112"); //20 "1"s + 1
    NumeroAstronomico* ceros1 = CrearDesdeCifraSeguidaDeCeros(1427, 90);
    NumeroAstronomico* ceros2 = CrearDesdeCifraSeguidaDeCeros(1, 70);
    NumeroAstronomico* aleatorio = CrearAleatorio();
    NumeroAstronomico* potenciar = CrearDesdeCadena("2");
    NumeroAstronomico* overflow = CrearDesdeCadena(
        "11111111111111111111111111111111"
        "1111111111111111111111111111111111111111111111111111"
        "1111111111111111111111111111111111111111111111111111"
        "11111111111111111111111111111111111111111111111111111111116"); // 203>100 digitos
    NumeroAstronomico* pre_overflow = CrearDesdeCifraSeguidaDeCeros(9, 99); // si lo sumo a si mismo desborda
    NumeroAstronomico* ceros_no_anda = CrearDesdeCifraSeguidaDeCeros(0, 14);

    NumeroAstronomico* temp_suma = NULL;

    //////// Algunos ejemplos ////////

    // PERSISTENCIA //

    // Escritura

    FILE* file_b = fopen("file_b.txt", "wb");
    FILE* file = fopen("file.txt", "w");

```

```
45      Write(file_b, largo1);
46      Write(file_b, chiquito);
47      Write(file_b, overflow);
48      Write(file_b, invalido);
49      Write(file_b, cosaNula);
50      Write(file_b, p1);
51
52      Print(file, largo1);
53      Print(file, chiquito);
54      Print(file, overflow);
55      Print(file, invalido);
56      Print(file, cosaNula);
57      Print(file, p1);
58
59      fclose(file_b);
60      fclose(file);
61
62      // Lectura
63
64      NumeroAstronomico* temp = CrearDesdeCadena(""); // para que tenga una
    inicializacion mas que malloc
65      NumeroAstronomico* total = CrearDesdeCadena("0");
66      NumeroAstronomico* total_b = CrearDesdeCadena("0");
67
68      printf("--------------Archivo NO binario--------------\n\n");
69      FILE* file_1 = fopen("file.txt", "r");
70      fseek(file_1, 0, SEEK_SET);
71      while(Scan(file_1, temp) != -1){
72          printf("Leido:\n");
73          Mostrar(temp, 3);
74          if (!EsError(temp)) {
75              temp_suma = Sumar(total,temp);
76              FreeNumeroAstronomico(total);
77              total = CrearDesdeCadena(temp_suma->entero);
78              FreeNumeroAstronomico(temp_suma);
79          }
80      }
81      printf("Suma total:\n");
82      Mostrar(total, 3);
83      fclose(file_1);
84
85      FreeNumeroAstronomico(temp);
86      temp = CrearDesdeCadena("");
87
88      printf("\n--------------Archivo binario--------------\n\n");
89      FILE* file_b_1 = fopen("file_b.txt", "rb");
90      fseek(file_b_1, 0, SEEK_SET);
91      while(Read(file_b_1, temp)){
92          printf("Leido:\n");
93          Mostrar(temp, 3);
94          if (!EsError(temp)) {
95              temp_suma = Sumar(total_b,temp);
96              FreeNumeroAstronomico(total_b);
97              total_b = CrearDesdeCadena(temp_suma->entero);
98              FreeNumeroAstronomico(temp_suma);
99          }
100     }
101     printf("Suma total:\n");
```

```
102     Mostrar(total_b, 3);
103     fclose(file_b_1);
104 
105     if(temp) FreeNumeroAstronomico(temp);
106     if(total) FreeNumeroAstronomico(total);
107     if(total_b) FreeNumeroAstronomico(total_b);
108 
109     // MOSTRAR //
110 
111     printf("p1, 3 grupos:\n"); Mostrar(p1, 3);
112     printf("p1_con_ceros, 3 grupos:\n"); Mostrar(p1_con_ceros, 3);
113     printf("p1, 5 grupos:\n"); Mostrar(p1, 5);
114     printf("p2, 2 grupos:\n"); Mostrar(p2, 2);
115     printf("p3, 4 grupos:\n"); Mostrar(p3, 4);
116     printf("chiquito, con 'exceso' de grupos "
117      "(se le piden 5 pero tiene solo 2 grupos para mostrar):\n"); Mostrar(
    chiquito, 5);
118     Mostrar(ceros1, 2);
119     Mostrar(ceros2, 3);
120     Mostrar(aleatorio, 4);
121     Mostrar(invalido, 4);     // Muestra error
122     Mostrar(overflow, 5);     // Muestra error
123     Mostrar(ceros_no_anda, 5);  // comportamiento indefinido
124 
125     // ERRORES //
126 
127     printf("%d\n", EsSecuenciaNula(cosaNula));                    // 1
128     printf("%d\n", EsSecuenciaInvalida(invalido));                // 1
129     printf("%d\n", EsSecuenciaInvalida(p3));                      // 0
130     printf("%d\n", p1->longitudError);                            // 26
131     printf("%d\n", EsError(chiquito) || EsError(cosaNula));       // 1
132 
133     // ARITMETICA //
134 
135     // Iguales
136     printf("%d\n", SonIguales(chiquito, otropeque)); // chiquito!=otropeque => 0
137     printf("%d\n", SonIguales(chiquito, pequenio)); // chiquito==pequenio => 1
138     printf("%d\n", SonIguales(p3, p3_equivalente)); // p3==p3_equivalente => 1
139     printf("%d\n", SonIguales(ceros1, ceros2));  // ceros1 > ceros2  => 0
140 
141     temp_suma = Sumar(cosaNula, chiquito);
142     printf("%d\n", SonIguales(chiquito, temp_suma)); // a+nulo = a => 1
143     FreeNumeroAstronomico(temp_suma);
144 
145     printf("%d\n", SonIguales(invalido, otro_invalido)); // mismo tipo de error
    => 1
146     printf("%d\n", SonIguales(p1, p1_con_ceros));   // son iguales => 1
147 
148     // Menor
149     printf("%d\n", EsMenor(p1, p2)); // p1<p2            => 1
150     printf("%d\n", EsMenor(p2, p1)); // p2>p1            => 0
151     printf("%d\n", EsMenor(p1, p1)); // p1==p1           => 0
152     printf("%d\n", EsMenor(largo1, largo2)); // largo1<largo2    => 1
153     printf("%d\n", EsMenor(largo1, largo1)); // largo1==largo1   => 0
154     printf("%d\n", EsMenor(largo2, largo1)); // largo2>largo1    => 0
155     printf("%d\n", EsMenor(ceros1, ceros2)); // ceros1 > ceros2  => 0
156     printf("%d\n", EsMenor(ceros2, ceros1)); // ceros2 < ceros1  => 1
157     printf("%d\n", EsMenor(aleatorio, ceros2)); // depende del azar
```

```
158
159     // Suma
160
161     // usando TEMP_SUMA porque si no salen leaks locos
162     // ver ASTRONUM.C : Sumar() para una explicacion (y una disculpa) del
163     // Guido del futuro (que ya es el Guido del pasado)
164
165     temp_suma = Sumar(p3, largo2);
166     Mostrar(temp_suma, 5);                                          // anda
167     FreeNumeroAstronomico(temp_suma);
168
169     temp_suma = Sumar(otropeque, ceros1);
170     Mostrar(temp_suma, 4);                                          // anda
171     FreeNumeroAstronomico(temp_suma);
172
173     for (int i=0; i<199; i++) {                                     // calcula
    2^200
174         temp_suma = Sumar(potenciar, potenciar);
175         FreeNumeroAstronomico(potenciar);
176         potenciar = CrearDesdeCadena(temp_suma->entero);
177         FreeNumeroAstronomico(temp_suma);
178     }
179
180     Mostrar(potenciar, 5); // muestra 2^200
181     Mostrar(pre_overflow, 4); // numero todo correcto
182
183     temp_suma = Sumar(pre_overflow, pre_overflow);
184     Mostrar(temp_suma, 4); // pero *2 da overflow (Muestra error)
185     FreeNumeroAstronomico(temp_suma);
186
187     temp_suma = Sumar(invalido, p1);
188     Mostrar(temp_suma, 5); // error de Sumar() y Mostrar(), se muestran
189     FreeNumeroAstronomico(temp_suma);
190
191     temp_suma = Sumar(cosaNula, cosaNula);
192     printf("%d\n", EsSecuenciaNula(temp_suma)); // nulo+nulo = nulo => 1
193     FreeNumeroAstronomico(temp_suma);
194
195     temp_suma = Sumar(cosaNula, p1);
196     Mostrar(temp_suma, 4); // nulo+p1 = p1
197     FreeNumeroAstronomico(temp_suma);
198
199     clock_t end = clock();
200     double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
201     printf("Tiempo de ejecucion: %f\n", time_spent);
202
203     // FREE POINTERS!!
204
205     FreeNumeroAstronomico(p1);
206     FreeNumeroAstronomico(p1_con_ceros);
207     FreeNumeroAstronomico(p2);
208     FreeNumeroAstronomico(p3);
209     FreeNumeroAstronomico(p3_equivalente);
210     FreeNumeroAstronomico(cosaNula);
211     FreeNumeroAstronomico(invalido);
212     FreeNumeroAstronomico(otro_invalido);
213     FreeNumeroAstronomico(chiquito);
214     FreeNumeroAstronomico(pequenio);
```

```
215     FreeNumeroAstronomico(otropeque);
216     FreeNumeroAstronomico(largo1);
217     FreeNumeroAstronomico(largo2);
218     FreeNumeroAstronomico(ceros1);
219     FreeNumeroAstronomico(ceros2);
220     FreeNumeroAstronomico(aleatorio);
221     FreeNumeroAstronomico(potenciar);
222     FreeNumeroAstronomico(overflow);
223     FreeNumeroAstronomico(pre_overflow);
224     FreeNumeroAstronomico(ceros_no_anda);
225 
226     return 0;
227 }
```

5.3. Archivo astronum.h

Este es el código del archivo de cabecera que contiene las directivas para el preprocesador para las definiciones de las estructuras utilizadas por el *TAD*, así como los prototipos de las funciones:

```
1  /////////// TAD ///////////
2  #ifndef TAD
3  #define TAD
4  
5  typedef struct {
6      char* entero; //tiene el numero en si
7      int longitudError;  //tiene el largo de la cadena, o el codigo de error
8  } NumeroAstronomico;
9  
10 typedef enum {           //codigo
11     //Ninguno,           //sin codigo, longitudError > 0
12     CadenaInvalida=-3,   // -3
13     Overflow,            // -2
14     PunteroNulo,         // -1
15     CadenaNula           //  0 (coincide con el largo en este caso)
16 } TipoDeError;
17 
18 #endif
19 
20 /////////// Creacion ///////////
21 #ifndef ASTRONUM_CREACION
22 #define ASTRONUM_CREACION
23 
24 NumeroAstronomico* CrearDesdeCadena(char* cadena);
25 NumeroAstronomico* CrearDesdeCifraSeguidaDeCeros(int num, int ceros);
26 NumeroAstronomico* CrearAleatorio();
27 void* FreeNumeroAstronomico(NumeroAstronomico* a);
28 
29 #endif
30 
31 /////////// Errores ///////////
32 #ifndef ASTRONUM_ERRORES
33 #define ASTRONUM_ERRORES
34 
35 int EsSecuenciaNula(NumeroAstronomico* a);
36 int EsSecuenciaInvalida(NumeroAstronomico* a);
37 int EsOverflow(NumeroAstronomico* a);
```

```
38 TipoDeError GetTipoDeError(NumeroAstronomico* a);
39 int EsError(NumeroAstronomico* a);
40 int EsPunteroNulo(NumeroAstronomico* a);
41
42 #endif
43
44 /////////////// Salida ////////////
45 #ifndef ASTRONUM_SALIDA
46 #define ASTRONUM_SALIDA
47
48 void Mostrar(NumeroAstronomico* a, int grupos);
49
50 #endif
51
52 /////////////// Aritmetica ////////////
53 #ifndef ASTRONUM_ARITMETICA
54 #define ASTROUNM_ARITMETICA
55
56 NumeroAstronomico* Sumar(NumeroAstronomico* a, NumeroAstronomico* b);
57 int SonIguales(NumeroAstronomico* a, NumeroAstronomico* b);
58 int EsMenor(NumeroAstronomico* a, NumeroAstronomico* b);
59
60 #endif
61
62 /////////////// Persistencia ////////////
63 #ifndef ASTRONUM_PERSISTENCIA
64 #define ASTRONUM_PERSISTENCIA
65
66 size_t Read(FILE* f, NumeroAstronomico* a);
67 void Write(FILE* f, NumeroAstronomico* a);
68 int Scan(FILE* f, NumeroAstronomico* a);
69 void Print(FILE* f, NumeroAstronomico* a);
70
71 #endif
```

5.4. Control de errores

Estas son las funciones que corresponden a la consulta de errores en nuestro TAD.

Son muy triviales, por lo que decidimos incluirlas en el Apéndice en lugar de en el desarrollo del documento.

Son en total cinco.

```
1     int EsSecuenciaNula(NumeroAstronomico* a){
2         return a->longitudError == CadenaNula;
3     }
4
5     int EsSecuenciaInvalida(NumeroAstronomico* a){
6         return a->longitudError == CadenaInvalida;
7     }
8
9     int EsOverflow(NumeroAstronomico* a){
10        return a->longitudError == Overflow;
11    }
12
13    int EsPunteroNulo(NumeroAstronomico* a){
14        return a->longitudError == PunteroNulo;
```

```
15      }
16
17      int EsError(NumeroAstronomico* a){
18          return EsSecuenciaInvalida(a) || EsOverflow(a) || EsSecuenciaNula(a) ||
        EsPunteroNulo(a);
19      }
```