



# 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

<b>1. Objetivos</b>	<b>2</b>
<b>2. Introducción</b>	<b>2</b>
<b>3. Implementación en ANSI C</b>	<b>2</b>
3.1. Definición del TAD . . . . .	3
3.2. Creación . . . . .	3
3.2.1. Crear desde cadena . . . . .	3
3.2.2. Crear desde entero seguido de ceros . . . . .	4
3.2.3. Crear aleatorio . . . . .	5
3.2.4. Liberar memoria . . . . .	6
3.3. Aritmética . . . . .	6
3.3.1. Comparaciones . . . . .	6
3.3.2. Suma . . . . .	7
3.4. Salida . . . . .	10
3.5. Persistencia . . . . .	12
3.5.1. Texto ( <b>Scan</b> y <b>Print</b> ) . . . . .	12
3.5.2. Binario ( <b>Read</b> y <b>Write</b> ) . . . . .	13
<b>4. Apéndice</b>	<b>15</b>
4.1. Repositorio en GitHub . . . . .	15
4.2. Ejemplos de uso . . . . .	15
4.3. Archivo <code>astronum.h</code> . . . . .	18
4.4. Control de errores . . . . .	19

## 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*<sup>1</sup> 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

<sup>2</sup>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 18.

---

<sup>1</sup>Disponible en C99

<sup>2</sup>Los comentarios en los programas no tendrán tildes en ninguna vocal, dado que esto causa errores a la hora de renderizar el documento en L<sup>A</sup>T<sub>E</sub>X.

### 3.1. Definición del TAD

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

```

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

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

```

1  typedef enum {
2      //Ninguno,           //sin codigo, longitudError > 0
3      CadenaInvalida=-3, // -3
4      Overflow,          // -2
5      PunteroNulo,       // -1
6      CadenaNula         // 0 (coincide con el largo en este caso)
7  } 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.

No haremos **malloc** para el puntero del entero, ya que copiaremos el puntero a la cadena que se pase como argumento.

```

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

```

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

```

1  for(int i=0; cadena[i]!='0'; *(cadena)++);

```

Si la cadena brindada solamente está compuesta por ceros, el comportamiento es indefinido.<sup>3</sup>

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

```

1      // Caso cadena muy larga (>100 caracteres)
2      if (strlen(cadena)>100) {
3          num->longitudError = Overflow;
4      }
5      // Caso todo bien (o cadena nula)
6      else if (cadenaNumerica(cadena)){
7          num->entero = cadena;
8          num->longitudError = strlen(cadena);
9      }
10     // Caso cadena invalida
11     else {
12         num->longitudError = CadenaInvalida;
13     }
14     return num;
15 }

```

En los casos de error, la cadena será descartada y el puntero `char* entero` contendrá información no definida.

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

```

1  // retorna (todos sus caracteres son numericos? 1 : 0)
2  static int cadenaNumerica(char* cadena){
3      for (int out=0; cadena[out]!='\0'; out++)
4          if (!isdigit(cadena[out])) return 0;
5      return 1;
6  }

```

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<sup>4</sup>, y luego retornará un `NumeroAstronomico*` creado con `CrearDesdeCadena(char*)` utilizando la cadena que generó.

<sup>3</sup>Como este *TAD* está definido para enteros positivos, para crear un Número Astronómico de valor nulo (sumable), se debe utilizar en su lugar la cadena nula.

<sup>4</sup>Si la cifra indicada es un cero, el comportamiento es indefinido.

Como precondition, 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 preconditiones
3      if (ceros>99) return NULL;
4      int longitud_num = (int) log10(num) + 1;    // cant de digitos de num
5      int longitud_total = longitud_num + ceros;
6      if (longitud_total > 100) return NULL;
7
8      // Creacion de la cadena
9      char* cadena = malloc(longitud_total);
10     if(cadena==NULL) return NULL; // fallo malloc
11     itoa(num, cadena, 10); // copia num a cadena
12     for(int i=0; i<ceros; i++) cadena[i+longitud_num]='0'; // llena con 0s
13     cadena[longitud_total] = '\0'; // fin de cadena
14
15     return CrearDesdeCadena(cadena);
16 }
```

### 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<sup>5</sup>.

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 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
15     return CrearDesdeCadena(cadena);
16 }
```

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:

---

<sup>5</sup>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 carácter de fin de cadena.

```

1 // Macros caracter a int y viceversa
2 #define can(c) (c-'0')
3 #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:

```

1 void* FreeNumeroAstronomico(NumeroAstronomico* a){
2     free(a->entero);
3     free(a);
4 }

```

## 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

Si tienen error, la cadena `entero` es indefinida así que se ignora, y si no tienen error y su longitud almacenada es distinta, ya podemos retornar 0 ya que ningún Número Astronómico comienza con 0.

Se implementa así:

```

1 int SonIguales(NumeroAstronomico* a, NumeroAstronomico* b){
2     if(EsError(a) && EsError(b))
3         return a->longitudError == b->longitudError;
4     if(a->longitudError != b->longitudError) return 0;
5     int largo = a->longitudError; // si llegan aca, son iguales en largo
6     int i=0;
7
8     for (i; ((i<largo) && (a->entero[i]==b->entero[i])); i++);
9     return i==largo;
10 }

```

Las funciones de control de errores (como `EsError(NumeroAstronomico*)`) están disponibles en la sección Apéndice, página 19.

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 caracteres 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->
3      longitudError);
4  }
```

### 3.3.2. Suma

Finalmente, la función `Suma(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.

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

Su código:



```

1  NumeroAstronomico* Sumar(NumeroAstronomico* a, NumeroAstronomico* b){
2      if(a==NULL || b==NULL) return NULL;
3      // Manejo de errores individuales
4      if (EsSecuenciaInvalida(a) || EsSecuenciaInvalida(b)){
5          fprintf(stderr, "Alguno de los numeros sumados es invalido.\n");
6          NumeroAstronomico* invalido = malloc(sizeof(NumeroAstronomico));
7          invalido->longitudError = CadenaInvalida;
8          return invalido;
9      }
10     if (EsSecuenciaNula(a) && !EsError(b))
11         return CrearDesdeCadena(b->entero); // se toma como sumar 0
12     if (EsSecuenciaNula(b) && !EsError(a))
13         return CrearDesdeCadena(a->entero); // lo mismo
14     if (EsSecuenciaNula(a) && EsSecuenciaNula(b))
15         return CrearDesdeCadena(""); // si los dos son nulos retorna nulo
16     // End manejo de errores individuales //
17
18     if (!EsError(a) && !EsError(b)){ // todo en orden
19         char* suma = sumaCadenas(a->entero, a->longitudError, b->entero, b->
longitudError);
20         // Si hubo Overflow (>100 caracteres), CrearDesdeCadena lo maneja
21         return CrearDesdeCadena(suma);
22     }
23     else { // algun error no manejado
24         fprintf(stderr, "Ocurrio un error.\n");
25         return NULL;
26     }
27 }

```

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<sup>6</sup> 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).

```

1  static char* sumaCadenas(const char* a, int lena, const char* b, int lenb){
2      // Longitud maxima, cadena larga, cadena corta
3      int longmax = lena > lenb ? lena : lenb;
4      longmax++;
5      char* corta = lena < lenb ? strdup(a) : strdup(b);
6      char* larga = lena < lenb ? strdup(b) : strdup(a);
7      char* salida = malloc(longmax+1); // lo que se retornara

```

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.

---

<sup>6</sup>Un 1 extra para el carácter de fin de cadena, y 1 extra para contemplar la longitud máxima de una suma de dos enteros.

Sumaremos comenzando en el segundo carácter, 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 “sobrestimado” 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?  $\rightarrow$  06300?
5. Tercera suma ( $2+2$ ): 06340?
6. Cuarta suma ( $7+7$  con carry): 06344?  $\rightarrow$  06354?
7. Fin de cadena: 06354\0
8. Arreglar puntero: 6354\0

Su implementación en C (con las Macros que definimos en la subsección 3.2 Creación pág. 5):

```

1 // Diferencia entre cadena de salida y corta (para rellenar con 0s)
2 int diff = longmax - strlen(corta);
3
4 // A sumar!
5 int buffer; // auxiliar para carries
6 int carries; // auxiliar para carries
7 salida[0] = '0'; // primera cifra en 0
8
9 // Suma caracter a caracter (de izquierda a derecha)
10 for(int i=1; i<longmax; i++){
11 // 0 o dígito (cadena corta)
12 char padding_or_num = i<diff? '0' : corta[i-diff];
13 // dígito (cadena larga)
14 int cifra = can(padding_or_num) + can(larga[i-1]);
15
16 // asigna el valor char de esa cifra (solo las unidades)
17 salida[i] = nac(cifra%10);
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 // El dígito N (carries) lugares a la izquierda + 1
23 buffer = can(salida[i-carries]+1);

```

```

24         // Se escribe en salida el valor de la unidad de eso
25         salida[i-carries] = nac(buffer%10);
26         carries++; // Por si el while itera
27     }
28 }
29
30 salida[longmax] = '\0'; // Fin de cadena
31 // Si empieza en 0 sobreestimamos, se arregla aqui
32 if (salida[0]=='0') *(salida)++;
33
34 return salida;
35 }

```

### 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:

```

1 void Mostrar(NumeroAstronomico* a, int grupos){
2     // Si hay algun error en el numero
3     // es mas facil detectarlo asi que usando las funciones EsTipoError() ...
4     if(a == NULL) { fprintf(stderr, "Puntero nulo\n"); return; }
5     switch(a->longitudError){
6         case Overflow: fprintf(stderr, "Overflow\n"); return;
7         case PunteroNulo: fprintf(stderr, "Puntero nulo\n"); return;
8         case CadenaInvalida: fprintf(stderr, "Cadena invalida\n"); return;
9         case CadenaNula: fprintf(stderr, "Cadena nula\n"); return;
10    }

```

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`:

```
1 #define MOSTRAR_GRUPOS_LINEAS 3
```

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

```
1 // Primer grupo, primera linea
2 int espacios = (3-(a->longitudError % 3)) % 3;
3 int i = 0-espacios;
4 for (i; i<3-espacios; i++)
5     (i<0)? putchar(' ') : putchar(a->entero[i]);
6 putchar('.');
7 // Terminar de mostrar primera linea
8 mostrarNGrupos(a, grupos-1, &i);
9 // Muestra el resto
10 // (indentado para que quede debajo del 2do grupo de la primera linea)
11 while(i < a->longitudError){
12     printf(" ");
13     mostrarNGrupos(a, MOSTRAR_GRUPOS_LINEAS, &i);
14 }
15 putchar('\n');
16 }
```

Dos cosas a notar:

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

```
1 #include <stdio.h>
2 int main(){
3     int a = 16;
4     int una = (3-(a%3))%3;
5     int otra = (-1*a)%3;
6     printf("Una: %d, Otra: %d\n", una, otra);
7     // Una: 2, Otra: -1
8     return 0;
9 }
```

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:

```
1 static void mostrarNGrupos(NumeroAstronomico* a, int cant, int *ind){
2     int tope = (*ind) + cant*3; //hasta que indice mostrar
3     int comienzo = (*ind); //en cual se empezo originalmente
4
5     for (*ind; (((*ind) < tope) && (a->entero[*ind] != '\0')); (*ind)++){
6         // este bucle cuenta la cantidad de digitos segun 'cant'
7         // y rompe si termino la cadena (!='\0')
8         putchar(a->entero[*ind]);
9         if ((*ind - comienzo)%3 == 2) putchar('.');
10    }
11    putchar('\n');
12 }
```

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:

`Scan`  $\iff$  `fscanf`  
`Print`  $\iff$  `fprintf`  
`Read`  $\iff$  `fread`  
`Write`  $\iff$  `fwrite`

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

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

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

Para conseguir eso, la función se implementa como sigue:

```

1 void Print(FILE* f, NumeroAstronomico* a){
2     if(a==NULL || f==NULL) return;
3     if(a->longitudError <= 0) //si tiene error
4         fprintf(f, "error %d\n", a->longitudError);
5     else
6         fprintf(f, "%s %d\n", a->entero, a->longitudError);
7 }
```

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):<sup>7</sup>

```

1 int Scan(FILE* f, NumeroAstronomico* a){
2     if(a==NULL || f==NULL) return 0;
3     // buffers
4     char cadena[100];
5     int longitud;
6     // lee longitud
7     int leido = fscanf(f, "%s %d\n", cadena, &longitud);
8     if(longitud > 0) //si no tiene error
9         a->entero = strdup(cadena);
10    a->longitudError = longitud;
11
12    return leido; // para conocer EOF
13 }
```

<sup>7</sup>No es necesario hacer un `free` porque se almacena en la pila.

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

```

1  #include <stdlib.h>
2  #include "astronum.h"
3
4  int main(){
5      FILE* f = fopen("file.txt", "r");
6      NumeroAstronomico* temp = malloc(sizeof(NumeroAstronomico));
7
8      while(Scan(f,temp) != -1){
9          Mostrar(temp, 3);
10     }
11     FreeNumeroAstronomico(temp);
12
13     return 0;
14 }
```

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 una cadena que contenga información no definida?

Su implementación:

```

1  void Write(FILE* f, NumeroAstronomico* a){
2      fwrite(&(a->longitudError), sizeof(int), 1, f);
3      if (a->longitudError > 0)
4          fwrite(a->entero, sizeof(char), 1+(a->longitudError), f);
5  }
```

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:

```

1  size_t Read(FILE* f, NumeroAstronomico* a){
2      // Primero lee longitud / error
3      int longitud;
4      size_t bytes_longitud = fread(&longitud, sizeof(int), 1, f);
5      size_t bytes_cadena = 0;
6      // Si no es error
7      if (longitud > 0){
8          char* cadena = malloc(longitud+1);
9          bytes_cadena = fread(cadena, sizeof(char), 1+longitud, f);
10         a->entero = cadena;
11     }
12     // Si es error, solo necesito el codigo de error
13     a->longitudError = longitud;
14
15     return bytes_cadena + bytes_longitud; //para conocer EOF
16 }
```

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:

```
1  #include <stdlib.h>
2  #include "astronum.h"
3
4  int main() {
5      FILE* f = fopen("file.txt", "rb");
6      NumeroAstronomico* temp = malloc(sizeof(NumeroAstronomico));
7
8      while(Read(f, temp)) {
9          Mostrar(temp, 3);
10     }
11     FreeNumeroAstronomico(temp);
12
13     return 0;
14 }
```

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

Con esto concluimos 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.





[illegible]

```

84     total_b = Sumar(total_b , temp);
85 }
86
87 printf("Suma total:\n");
88 Mostrar(total_b , 3);
89 fclose(file_b_1);
90
91 if(temp!=NULL)
92     FreeNumeroAstronomico(temp);
93 if(total!=NULL)
94     FreeNumeroAstronomico(total);
95 if(total_b!=NULL)
96     FreeNumeroAstronomico(total_b);
97
98 // MOSTRAR //
99
100 printf("p1, 3 grupos:\n"); Mostrar(p1, 3);
101 printf("p1_con_ceros, 3 grupos:\n"); Mostrar(p1_con_ceros , 3);
102 printf("p1, 5 grupos:\n"); Mostrar(p1, 5);
103 printf("p2, 2 grupos:\n"); Mostrar(p2, 2);
104 printf("p3, 4 grupos:\n"); Mostrar(p3, 4);
105 printf("chiquito, con 'exceso' de grupos "
106 " (se le piden 5 pero tiene solo 2 grupos para mostrar):\n"); Mostrar(
chiquito , 5);
107 Mostrar(ceros1 , 2);
108 Mostrar(ceros2 , 3);
109 Mostrar(aleatorio , 4);
110 Mostrar(invalido , 4); // Muestra error
111 Mostrar(overflow , 5); // Muestra error
112 Mostrar(ceros_no_anda , 5); // comportamiento indefinido
113
114 // ERRORES //
115
116 printf(" %d\n", EsSecuenciaNula(cosaNula)); // 1
117 printf(" %d\n", EsSecuenciaInvalida(invalido)); // 1
118 printf(" %d\n", EsSecuenciaInvalida(p3)); // 0
119 printf(" %d\n", p1->longitudError); // 26
120 printf(" %d\n", EsError(chiquito) || EsError(cosaNula)); // 1
121
122 // ARITMETICA //
123
124 // Iguales
125 printf(" %d\n", SonIguales(chiquito , otropeque)); // chiquito!=otropeque =>
0
126 printf(" %d\n", SonIguales(chiquito , pequeno)); // chiquito==pequeno => 1
127 printf(" %d\n", SonIguales(p3 , p3_equivalente)); // p3==p3_equivalente => 1
128 printf(" %d\n", SonIguales(ceros1 , ceros2)); // ceros1 > ceros2 => 0
129 printf(" %d\n", SonIguales(chiquito , Sumar(cosaNula , chiquito))); // a+nulo
= a => 1
130 printf(" %d\n", SonIguales(invalido , otro_invalido)); // mismo tipo de error
=> 1
131 printf(" %d\n", SonIguales(p1 , p1_con_ceros)); // son iguales => 1
132
133 // Menor
134 printf(" %d\n", EsMenor(p1 , p2)); // p1<p2 => 1
135 printf(" %d\n", EsMenor(p2 , p1)); // p2>p1 => 0
136 printf(" %d\n", EsMenor(p1 , p1)); // p1==p1 => 0
137 printf(" %d\n", EsMenor(largo1 , largo2)); // largo1<largo2 => 1

```

```

138 printf("%d\n", EsMenor(largo1, largo1)); // largo1==largo1 => 0
139 printf("%d\n", EsMenor(largo2, largo1)); // largo2>largo1 => 0
140 printf("%d\n", EsMenor(ceros1, ceros2)); // ceros1 > ceros2 => 0
141 printf("%d\n", EsMenor(ceros2, ceros1)); // ceros2 < ceros1 => 1
142 printf("%d\n", EsMenor(aleatorio, ceros2)); // depende del azar
143
144 // Suma
145 Mostrar(Sumar(p3, largo2), 5); // anda
146 Mostrar(Sumar(otropeque, ceros1), 4); // anda
147 for(int i=0; i<199; i++)
148     potenciar = Sumar(potenciar, potenciar); // calcula 2^200
149 Mostrar(potenciar, 5); // muestra 2^200
150 Mostrar(pre_overflow, 4); // numero todo correcto
151 Mostrar(Sumar(pre_overflow, pre_overflow), 4); // pero *2 da overflow (
Muestra error)
152 Mostrar(Sumar(invalido, p1), 5); // error de Sumar() y Mostrar(), se
muestran
153 printf("%d\n", EsSecuenciaNula(Sumar(cosaNula, cosaNula))); // nulo+nulo =
nulo => 1
154 Mostrar(Sumar(cosaNula, p1), 4); // nulo+p1 = p1
155
156 clock_t end = clock();
157 double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
158 printf("Tiempo de ejecucion: %f\n", time_spent);
159
160 return 0;
161 }

```

### 4.3. Archivo astronom.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 {
11     //Ninguno, //codigo
12     CadenaInvalida=-3, //sin codigo, longitudError > 0
13     Overflow, // -3
14     PunteroNulo, // -2
15     CadenaNula // -1
16 } TipoDeError; // 0 (coincide con el largo en este caso)
17
18 #endif
19
20 /////////////// Creacion ///////////////////
21 #ifndef ASTRONUMCREACION
22 #define ASTRONUMCREACION
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 ASTRONUMERRORES
33 #define ASTRONUMERRORES
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 ASTRONUMSALIDA
46 #define ASTRONUMSALIDA
47
48 void Mostrar(NumeroAstronomico* a, int grupos);
49
50 #endif
51
52 /////////////// Aritmetica ///////////////////
53 #ifndef ASTRONUMARITMETICA
54 #define ASTROUNMLARITMETICA
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

```

#### 4.4. 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) ||
19     EsPunteroNulo(a);
20 }
```