Introducción:

El objetivo fundamental del trabajo es implementar una calculadora de números enteros de precisión fija, en general arbitrariamente grande. La precisión es la cantidad de dígitos decimales que conforman a un número. Se deberán usar "listas" para almacenar los dígitos de los números.
Además se agregara un manejo de archivos de entrada de instrucciones y de salida de resultados, leyendo las rutas de archivos por la línea de comandos.
Por último se deberá probar que el programa no tiene pérdidas de memoria.

Concepto de listas:

Una lista es una estructura de datos formada por elementos de un mismo tipo, llamados nodos. Dependiendo del tipo de lista, la estructura del nodo estará pensada para almacenar cierto tipo de información. Lo que todas las listas tienen en común es que los nodos siempre poseen un puntero al nodo próximo de la estructura.
Cuando la lista es doblemente enlazada, los nodos además poseen un puntero al nodo anterior de la lista, permitiendo recorrer la lista bidireccionalmente. Los elementos almacenados en una lista pueden ser agregados, borrados y accedidos sin restricciones, en cualquier punto de la estructura.
Cuando un nodo es el primero de la lista, se indica con la dirección 'NULL' en su puntero al nodo anterior. Cuando el nodo es el último de una lista, se pone NULL en su puntero al siguiente nodo.

Diseño e implementación del programa:

- ❏ Acerca de las operaciones matemáticas:

El primer paso al realizar una operación matemática en el caso de una suma o resta, es fijarse en los signos de los operandos para detectar por ejemplo, una resta camuflada en una suma, como es el caso de 8+(-9) , que sería lo mismo que 8-9 .
La regla que encontramos es que si los signos de los operandos son opuestos, se debe realizar la operación contraria (si es suma, restar), y que el signo del primer operando determina si el resultado de la operación (resta o suma) de los módulos, se debe cambiar de signo.

Para sumar simplemente usamos un algoritmo que suma digito por digito y usa un acarreo.

Para restar comparamos el módulo de los dos operandos para siempre restar al operando más grande el operando más chico, y así simplificar el resto del

procedimiento. Realizamos una resta dígito por dígito usando un borrow (método casi idéntico a la suma), y por último calculamos el signo.

Para el producto hacemos el producto de todo el primer operando, por una cifra del segundo operando, y así para todas las cifras. Estos productos intermedios se multiplican por potencias de 10 y se los va sumando, usando la función de add_bignum. Por último calculamos el signo, que es positivo a menos que los operandos tuvieran signos diferentes (similar a XOR).

- ❏ Acerca de las listas:

Incluimos un archivo "listas.c" el cual tiene las funciones para obtener el nodo número x de una lista, el último nodo de una lista, para agregar nodos al inicio o al final, que luego utilizamos . Las listas simplificaron una función para añadir ceros a la izquierda de un número (ejemplo f(123) = 000123) , ya que cuando esto se implementaba con arreglos, había que desplazar de lugar cada dígito.

- ❏ Acerca de los archivos:

Para la salida de archivos usamos basicamente fprintf(archivo,texto) , y el parametro archivo podía apuntar a un archivo real, o ser stdout, por lo que fue fácil. En cambio para la entrada de archivos utilizamos fscanf si feof==0 , y para teclado scanf.

Interfaz de línea de comandos

El programa se podrá ejecutar en distintos modos, colocando parámetros en la línea de comandos del terminal. La sintaxis a utilizar es la siguiente:

./main.exe modo -input input.txt -output output.txt -p 1000

Los parámetros disponibles son los siguientes (se puede intercambiar su orden también):

- **modo**:
  - **superCalc** : Calculadora de precisión arbritaria (el programa del TP n°3 pero basicamente usando listas).
  - **simpleCalc** : Calculadora de ocho operaciones básicas (del TP n°1) y graficador (del TP n°2)
- **-input input.txt** : Donde "input.txt" es un archivo en el directorio del ejecutable (./) y se puede cambiar su nombre y/o poner una ruta absoluta como "/home/usuario/carpeta/input.in". Indica que el archivo será la entrada de operaciones para superCalc. Un ejemplo de archivo válido sería "3+4\n#calculate\n" (con \n salto de línea).

- **-output output.txt** : Igual que en input, el nombre "output.txt" es un ejemplo. Con este parámetro indicamos un archivo para, en modo superCalc guardar los resultados, o en modo graficador guardar un gráfico.
- **-p 1000** : Donde "1000" puede ser cualquier número natural, indica la precisión de superCalc.

Ejemplos:

- **./main.exe**
  El programa muestra un menú donde manualmente podemos ejecutar una función de la calculadora simple, el graficador, o la calculadora de precisión arbitraria.

- **./main.exe simpleCalc**
  El programa se ejecuta igual que en el primer ejemplo.

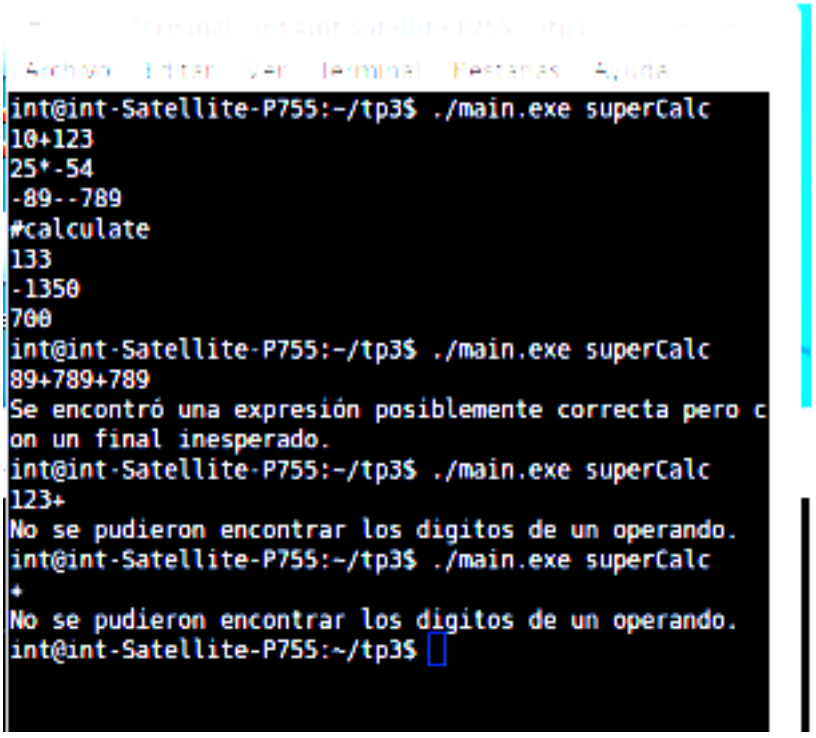
- **./main.exe superCalc -p 500**
  El programa entra directamente al modo superCalc, esperando que se ingresen las cadenas de números para operar. Agregando a la derecha de "superCalc", "-p 500", pedimos que la precisión del resultado sea de 500 (también puede ser otro número). Si no se añade esto último, la precisión es 100.

- **./main.exe superCalc -input big.in -output big2.out**
  El programa entra directamente al modo superCalc, y en lugar de recibir operaciones por teclado, el sistema operativo se "las escribe", copiandolas del contenido del archivo big.in, y guarda los resultados que se imprimirían por pantalla, en el archivo big2.out.
  Los símbolos < y > son controles de flujo, de entrada stdin y de salida stdout. El sistema operativo es el único que se encarga de archivos y el que se entera de que se introdujeron esos parámetros.

- **./main.exe simpleCalc -output grafico.pbm**
  Con esto podemos guardar un gráfico hecho con el graficador, en el archivo grafico.pbm

Ejecuciones de prueba

Primero probamos algunas operaciones como ejemplo y el programa que verifica las expresiones matemáticas:



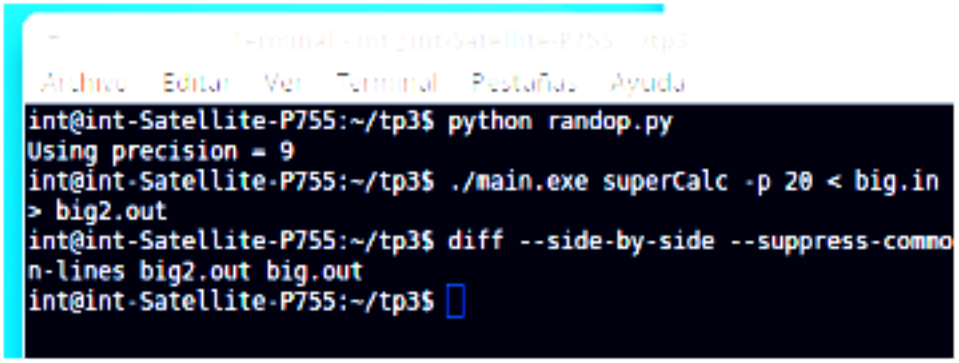
Luego podemos probar que los cálculos se realizan correctamente ejecutando lo siguiente:

```
python randop.py
main.exe superCalc -p 8 -input big.in -output big2.out
diff --side-by-side --suppress-common-lines big2.out big.out
```

Esto ejecuta el programa randop.py , que genera un archivo de operaciones “big.in” y un archivo con los resultados de las operaciones “big.out”.

Luego se ejecuta nuestro programa superCalc con precisión de ocho dígitos y empieza a calcular las mismas operaciones del archivo big.in , guardando los resultados en big2.out .

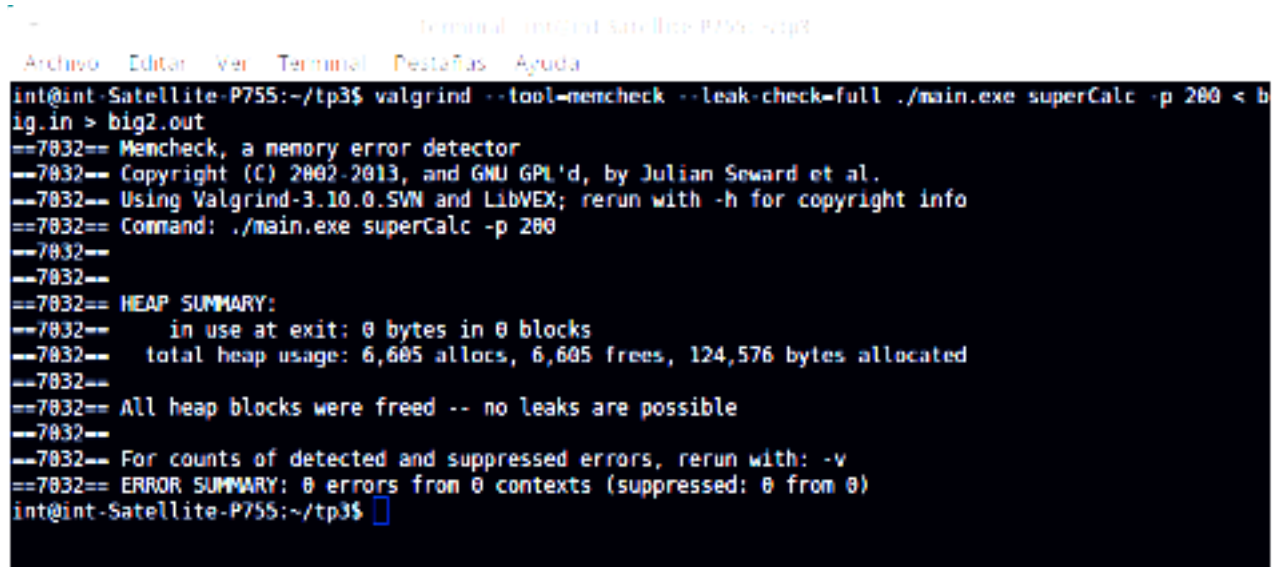
Por último, debemos verificar si los archivos de resultados son iguales. Para esto usamos el programa diff , que imprimiría las líneas de texto que no sean idénticas entre los archivos big.out y big2.out . Por lo tanto si termina la ejecución sin imprimir ningún mensaje, salvo “Infinito / Overflow” cuando el resultado tiene más de ocho dígitos, el programa debería estar funcionando correctamente.



Ahora podemos probar que el programa no posea pérdidas de memoria. Teniendo un archivo big.in con varias operaciones de distintos tipos, ejecutamos lo siguiente:

```
valgrind --tool=memcheck --leak-check=full ./main.exe superCalc -p 200 -input big.in -output big2.out
```

Las líneas "no leaks are possible" y "ERROR SUMMARY: 0 errors from 0 contexts", nos indican que no hay escrituras en memoria inválida (como las escrituras fuera de un arreglo) y que no hay pérdidas de memoria.



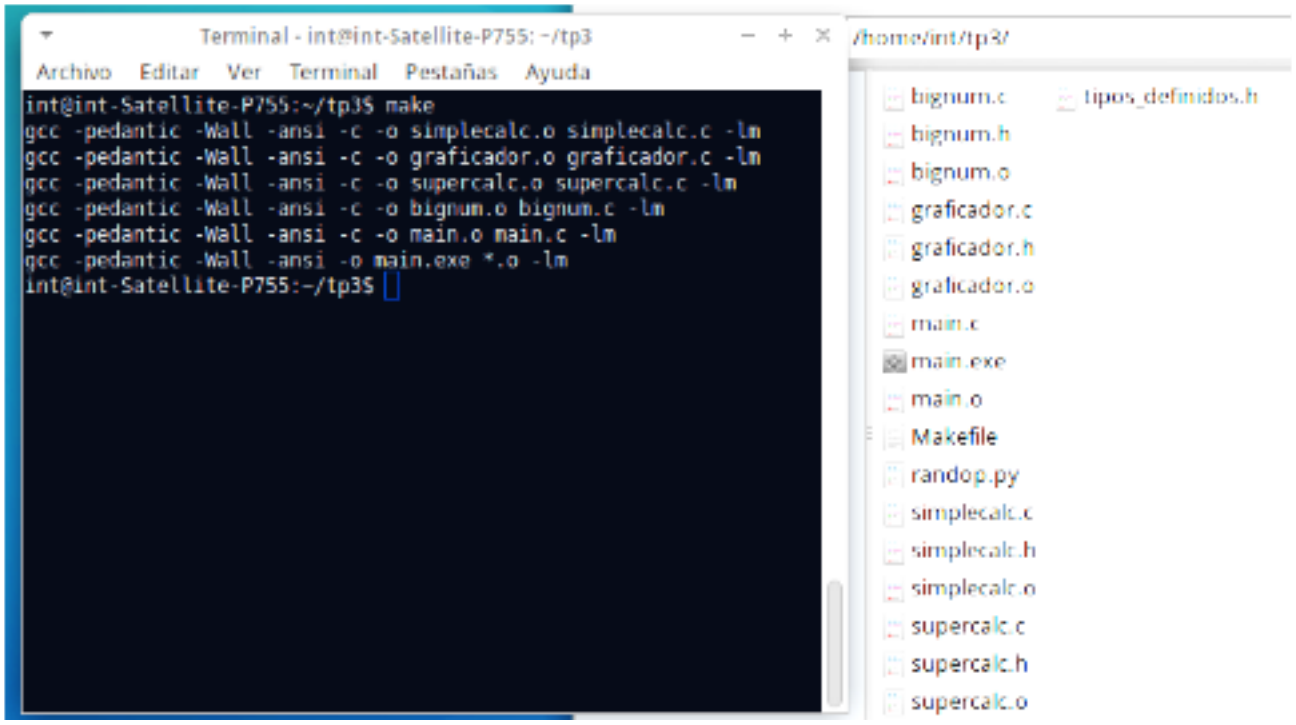
Proceso de compilación

Creación del ejecutable a partir del código fuente:

Se deben tener todos los archivos de código *.c, *.h y el Makefile en una misma carpeta.
En el terminal entramos a la carpeta usando el comando “cd” como en los trabajos anteriores, para luego ejecutar “make” . El programa make interpretará el archivo “Makefile” y ejecutará la línea “all”, que será equivalente a ejecutar lo siguiente en un terminal:

```
gcc -pedantic -Wall -ansi -c -o bignum.o bignum.c -lm
gcc -pedantic -Wall -ansi -c -o graficador.o graficador.c -lm
gcc -pedantic -Wall -ansi -c -o simplecalc.o simplecalc.c -lm
gcc -pedantic -Wall -ansi -c -o supercalc.o supercalc.c -lm
gcc -pedantic -Wall -ansi -c -o listas.o listas.c -lm
gcc -pedantic -Wall -ansi -c -o main.o main.c -lm
gcc -pedantic -Wall -ansi -o main.exe main.c *.o -lm
```

Si queremos borrar los archivos compilados y volver a tener solamente el código fuente, podemos ejecutar “make clean”, que eliminará los archivos objeto y los ejecutables de la carpeta (todos los .o y .exe).



Como se observa en esta imágen, indroducir “make” basta para compilar el código en el ejecutable main.exe .