Documentación Proyecto #1 Algoritmos y Estructuras de Datos 2

Contenidos:

- Breve descripción del problema.
- Diagrama de clases.
- Descripción de las estructuras de datos desarrolladas.
- Descripción detallada de los algoritmos desasrrollados.

Breve descripción del Problema

Consiste en el diseño e implementación de un IDE para el pseudo lenguaje C!. El IDE debe presentar un layout que permita ver el áre del código, el conteo de líneas, botón para correr, consola y un log donde se registren los movimientos y errores y un área que permita ver las variables, su localización en memoria, la cantidad de referencias a esta misma y su valor. El pseudo lenguaje C! debe tener una sintaxis derivada del lenguaje C y también se asumen reglas iguales a las de este lenguaje. En la consola se debe imprimir todo lo que represente una llamada a esta misma con algún comando elegido por los estudiante, y en el log cualquier error interno, llamadas al servidor de memoria junto con lo que ocurre internamente. En el RAM live view se debe observar en tiempo real el estado de la RAM para saber de cada variable su nombre, valor, dirección de memoria y número de referencias.

El lenguaje C! posee los tipos de datos:

- int
- long
- char
- float
- double
- struct (datos primitivos, no soporta anidación)

Permite solamente un "scope" mediante { }, para deliitar la visibilidad de las variables que se encuentren dentro de los { }. Una de fuera se puede acceder, pero acceder desde afuera una de adentro no es posible.

Respecto a la memoria, se maneja desde un servidor aparte, que se conectaría con el IDE (el cuál funcionaría como cliente), por medio de los sockets y este realiza un único malloc de la memoria total y luego escucha peticiones y se comunica con el IDE por medio de un objeto JSON y en este se realizan todos los procesos relacionados al manejo de la memoria. Se lleva conteo de referencias y cuando una variable no es referenciada, se elimina en el garbage collector.

Descripción de las estructuras de datos desarrolladas

Se utilizan listas enlazadas para distintos fines en este proyecto, como textos y memoria, además de que cada una posee funcionalidades aparte que le permiten integrarse mejor al programa y facilitar muchas funciones para interactuar con esta misma.

Code TxT

Una de estas es la de la clase Code TxT, esta lista se encara de guardar todos los textos en la interfaz, se crea una distinta para cada uno: cada columna del RAM Viewer, el log y la consola, y también el código y número de líneas. Los nodos de la lista corresponden a un objeto denominado línea, ya que en un inicio se planeaba utilizar solamente para el código, pero luego gracias a todas las funciones que se incorporaron después, se decidió usar esta misma lista para todo lo que requiriera texto, ya que permite mover el texto según los límites del espacio en el que se encuentre y también lleva conteo de cada línea, por lo que utilizar el mismo objeto como base para la lógica de todos los textos facilitaba bastante más el trabajo. También recalcar, que si hay ciertos métodos incluidos con funcionalidades exclusivas para el código, ya que si era necesario que el texto del código tuviera cierto comportamiento según el caso. Un claro ejemplo de esto es el método para enviar la información, que se basa en un atributo que mantiene su valor como el primer elemento de la lista, y conforme se da click en el botón de next, para avanzar en como se corre el código, este va avanzando al siguiente nodo, y de esta manera logra enviar al server cada línea del código por separado y que se ejecute también de esta forma. Los demás textos que emplean esta lista solamente utilizan su método Draw, que se encarga de colocar cada texto de cada línea en la

pantalla, según la posición que cada línea tenga definida. Cada vez que se inserta una línea nueva, se aumentan los valores de las posiciones para el texto y se aumenta también el contador de líneas de la lista. No hay un método para borrar, ya que se reutiliza el espacio, el nodo queda y luego se modifica su valor, pero no se borran las líneas en ningún momento, a menos que se resetee el objeto, el cual devuelve la primera posición a un valor de nullptr.

AvaiList

La estructura anterior es una lista doblemente enlazada con cabeza y cola que es usada para almacenar los 10 megas de memoria que se utilizarán durante la ejecución del código escrito en el IDE AtomiC!.

La lista en cuestión permite lo siguiente:

1. Solicitar el primer nodo y desenlazarlo de la lista con el fin de liberar su memoria o utilizar la memoria en otro fin.
2. Obtener el el nodo ubicado en la posición del índice. Como añadido, soporta introducir índices negativos que hacen un conteo regresivo que comienza en la cola y se dirige hacia la cabeza.
3. Pedir el primer o último elemento.

MemoryList

Esta estructura es una lista doblemente enlazada con cabeza y cola que hereda de la clase AvaiList. Es usada para almacenar la memoria que se encuentre utilizada durante la ejecución del código.

Esta estructura incluye, además de las que incluye AvaiList, las siguientes:

1. Cambiar el valor de un nodo buscado por su nombre.
2. Obtener un nodo mediante referencia o mediante llamada normal.
3. Obtener el tamaño de la lista.

Algoritmos Desarrollados

Interfaz y Sockets

En el programa se hace uso de SFML para la creación de la interfaz, la cual se debe ejecutar mediante un thread para que de esta manera funcione también la conexión con sockets con el servidor, ya que ambos requieren de un ciclo while que mantiene su ejecución hasta que se detenga el programa. Para hacer esto posible se emplea la biblioteca thread, para así ejecutar a la vez el servidor y la interfaz.

El IDE actua como el cliente, por lo que para que este permita que el programa se siga corriendo, debe poder establecer conexión con el servidor, si este no se está ejecutando, el cliente termina su ejecución con un exit code 1. Si logra establecer la conexión, se inicia un ciclo while que se mantiene a la espera de cualquier mensaje que reciba proveniente del servidor, ya sean instrucciones, objetos de tipo JSON o errores notificados por el servidor, y se comunica con la interfaz para que realice algún proceso según sea el caso.

La interfaz, como ya se indició, se mantiene abierta mediante un ciclo while. Este ciclo se encarga de ir colocando en pantalla todos los botones y objetos en cada frame que se actualiza, y para cambiar valores, se actualiza propiamente los valores de los objetos, ya sean textos, botones o propiamente el cuadro de texto en el cual se introduce el código. En el mismo ciclo while se reciben los eventos tales como clicks, movimiento del mouse y cualquier tecla que se presione en el teclado, y realizar algún proceso según el evento que detecte.

Parseo de código e intérprete.

Para la manipulación del código ingresado por el usuario, el algoritmo implementado busca principalmente las palabras clave: “int”, “long”, “float”, “double”, “char” o “struct”. Una vez encontrado, el texto que procede, corresponderá al nombre de la variable. Al lado contrario de la igualdad (`=`) se encuentran los datos.

- Todos los datos obtenidos en code parser, son manipulados por el administrador de memoria expuesto más adelante.
- Si el dato es de tipo `int` o `long` y se ingresa cualquier cosa que no sea una referencia o una variable o un número entero específicamente, el código terminará y mostrará un error en el Log.

- De otro modo, si el dato es de tipo `float `o `double`, el dato que se almacena tendrá coma flotante. Se aceptan números enteros en la declaración, sin embargo, se almacenará como de punto flotante.
- Un dato de tipo `char` es un dato único en C!, pues no es una variable numérica, ni una referencia, es un caracter. El dato requiere comillas simples (`'`) alrededor del caracter (`'a'`, por ejemplo).
- Los struct son meramente referencias a bloques de código con instrucciones dentro.

Manipulación y administración de memoria.

El administrador de memoria es una instancia única que incluye dos listas doblemente enlazadas para el almacenamiento de la memoria.

Para la reserva de memoria se realiza un “malloc” de 10 * 2^20 bits o 10 MB y dividir la memoria reservada en 131 072 punteros de 8 bits de tipo Nodo, donde se almacenarán los datos.

- Available Memory: almacena 131 072 nodos vacíos con memoria disponible para utilizar.
- Memory List: almacena la memoria que está en uso. En esta memoria se almacenan nodos con el nombre de las variables, el valor como un puntero de tipo void, el tipo de variable que se está manipulando y el nombre de la clase a la que pertenece.

Al guardar un dato nuevo, se ingresa el nombre de la variable, la clase a la que pertenece y el valor se recibe como un puntero vacío.

Para almacenar el dato, previamente se realiza un casting hacia “void” de la siguiente manera: “void temp = void” new int(intVal); cuando intVal es una variable de tipo “int” que almacena temporalmente el valor ingresado por el usuario.