Proyecto I – VSCode Memory Manager

Instituto Tecnológico de Costa Rica
Área de Ingeniería en Computadores
Algoritmos y Estructuras de Datos II (CE 2103)
Primer Semestre 2020
Valor 25%

Objetivo General

- ➔ Diseñar e implementar una extensión de Visual Studio Code para manejo de la memoria.

Objetivos Específicos

- ➔ Investigar y desarrollar una aplicación en el lenguaje de programación C++
- ➔ Investigar acerca de programación orientada a objetos en C++.
- ➔ Aplicar sobrecarga de Operadores en C++
- ➔ Implementar arquitectura cliente - servidor.

Descripción del Problema

Para facilitar el uso de manejo de memoria de los programadores en C++ se propone crear una extensión para el IDE Visual Studio Code, esta va a servir como una herramienta para facilitar el manejo de memoria con funcionalidades como Heap Visualizer, Garbage Collector y Remote Memory.

Arquitectura de la aplicación:

La aplicación debe funcionar por medio de una extensión de Visual Studio Code cuyo rol es ilustrar de una manera cómoda al usuario todas las posibilidades de la aplicación, para aplicar sus funcionalidades debe de llamar a una biblioteca dinámica de C++, donde va a manejar la memoria y aplicar los algoritmos que la aplicación tiene que hacer, es decir, la extensión solo servirá de interfaz.

VSPointer

La biblioteca va a funcionar mediante la creación de un nuevo tipo de dato denominado VSPtr<T>, este tiene al menos un atributo de tipo T*, se instancia de la siguiente manera:

```
VSPtr<int> myPtr = VSPtr<int>::New();
```

En este caso, aparte de reservar la memoria para VSPtr, también se reserva la memoria para almacenar un int. Posteriormente, el programador puede almacenar el dato deseado utilizando:

```
*myPtr = 5.
```

En este caso, el operador * es sobrecargado y se almacena el valor 5 en el espacio reservado para un int. De la misma forma, si se desea obtener el valor guardado en myPtr, el programador haría lo siguiente:

```
int valor = &myPtr;
```

El operador = debe verificar el tipo de los operandos. Si ambos son VSPointer, copia el puntero y el id de VSPointer dentro del Garbage Collector (más sobre esto adelante).

```
VSPtr<int> myPtr = VSPtr<int>::New();
VSPtr<int> myPtr2 = VSPtr<int>::New();
*myPtr = 5.
myPtr2 = myPtr; //Deja en myPtr2 una referencia a la misma dirección de memoria donde está 5
```

```
y copia el id.
```

Ahora bien, si el operando de la derecha es de tipo distinto a VSPointer, se verifica si el tipo es el mismo a lo interno de VSPointer. De ser así se guarda el valor en el puntero interno:

```
myPtr = 6 // es equivalente a *myPtr = 6
```

El operador & se sobrecarga y se obtiene el valor guardado. VSPointer tiene un destructor que se encarga de liberar la memoria del pointer guardado internamente.

Garbage Collector

Al llamarse la biblioteca se debe inicializar la clase singleton Garbage Collector que se ejecuta como un thread cada n segundos. Cada vez que se llama el método New de VSPointer, se guarda la dirección de memoria de la instancia de VSPointer dentro de la clase del Garbage Collector.

El Garbage Collector va a guardar las direcciones de memoria de cada VSPointer conocido, y le da a cada instancia un ID autogenerado. El destructor de VSPtr llama al Garbage Collector para indicar que la referencia se ha destruido. Una vez que el conteo de referencias de un VSPointer llegue a cero, el Garbage Collector lo libera, evitando memory leaks.

Heap Visualizer e Interfaz

Esta funcionalidad le permite al usuario por medio de la interfaz de Visual Studio Code visualizar cómo se está administrando la memoria. Va de lado con el Garbage Collector, debido a que muestra lo que almacena el Garbage Collector. Para esto se debe mostrar el tipo de dato, el valor del dato, la ubicación en memoria y el número referencias de cada VSPtr almacenado en el Garbage Collector.

Además le va a proveer al usuario la opción de conectarse un servidor remoto, para guardar los punteros en el heap del servidor, en lugar de la memoria de la computadora, la interfaz debe mostrar cuando se está usando la memoria del servidor o la de la computadora.

La interfaz permite al usuario insertar la IP del servidor, una contraseña que protege al servidor, el puerto donde va a estar ubicado el servidor y un nombre que el usuario desee para la conexión, junto a un botón para probar la conexión. Al tocar el botón si la conexión es exitosa, guarda la conexión dentro de un archivo json.

La aplicación va a tener un comando en el Command Palette (Activable mediante control+shift+p en Linux y Windows o cmd + shift + p en MacOS) de Visual Studio Code que va a crear una copia de la biblioteca dinámica en el proyecto y la va a importar en el main del proyecto. Además de habilitar al usuario el uso de todas las herramientas de la extensión, es decir, hasta que esta opción no se use, toda la extensión se va a encontrar deshabilitada.

Remote Memory

Este opción se activa mediante la interfaz. Esta funcionalidad le da la opción a un usuario de guardar los datos en un servidor, para esto el servidor va a tener una contraseña, una dirección IP y un puerto. La contraseña va a tener que ser encriptada mediante el algoritmo MD5.

Al instanciar los VSPointer estos van a ser enviados al servidor mediante un json que le va a indicar el tipo y el dato y le retorna a la biblioteca un ID, que va identificar el dato, cada vez que el usuario pida usar ese dato, la biblioteca va a enviar el ID al Servidor, y este le va a responder con un Json con todo lo necesario para usar el dato . El servidor va a contar con su propio Garbage Collector que funciona igual que el de la biblioteca, y el servidor puede almacenar información de varios Usuarios, es decir, varias instancias de la biblioteca pueden ser usadas a la vez desde diferentes computadoras en diferentes proyectos y se pueden conectar al mismo servidor.

Documentación requerida

1. Internamente, el código se debe documentar utilizando DoxyGen y se debe generar el HTML de la documentación.
2. Dado que el código se deberá mantener en GitHub, la documentación externa se hará en el Wiki de GitHub. El Wiki deberá incluir:
    a. Breve descripción del problema
    b. **Planificación y administración del proyecto:** se utilizará la parte de project management de GitHub para la administración de proyecto. Debe incluir:
        ➔ Lista de features e historias de usuario identificados de la especificación
        ➔ Distribución de historias de usuario por criticalidad
        ➔ Plan de iteraciones que agrupen cada bloque de historias de usuario de forma que se vea un desarrollo incremental
        ➔ Descomposición de cada user story en tareas.
        ➔ Asignación de tareas a cada miembro del equipo.
    c. Diagrama de clases en formato JPEG o PNG
    d. Descripción de las estructuras de datos desarrolladas.
    e. Descripción detallada de los algoritmos desarrollados.
    f. Problemas encontrados en forma de bugs de *github*: En esta sección se detalla cualquier problema que no se ha podido solucionar en el trabajo.

Aspectos operativos y evaluación:

1. **Fecha de entrega: De acuerdo al cronograma del curso**
2. El proyecto tiene un valor de 25% de la nota del curso.
3. El trabajo es en **parejas.**
4. Es obligatorio utilizar un GitHUB.
5. Es obligatorio integrar toda la solución.
6. El código tendrá un valor total de 85%, la documentación 15%.
7. De las notas mencionadas en el punto anterior se calculará la Nota Final del Proyecto.
8. Se evaluará que la documentación sea coherente, acorde a la dificultad/tamaño del proyecto y el trabajo realizado, se recomienda que realicen la documentación conforme se implementa el código.
9. La nota de la documentación es proporcional a la completitud del proyecto.
10. La documentación se revisará según el día de entrega en el cronograma.
11. Las citas de revisión oficiales serán determinadas por el profesor durante las lecciones o mediante algún medio electrónico.
12. Los estudiantes pueden seguir trabajando en el código hasta 15 minutos antes de la cita revisión oficial
13. Aún cuando el código y la documentación tienen sus notas por separado, se aplican las siguientes restricciones

a. Si no se entrega documentación, automáticamente se obtiene una nota de 0.
b. Si no se utiliza un manejador de código se obtiene una nota de 0.
c. Si la documentación no se entregan en la fecha indicada se obtiene una nota de 0.
d. Sí el código no compila se obtendrá una nota de 0, por lo cual se recomienda realizar la defensa con un código funcional.
e. El código debe ser desarrollado en C++ (Linux), en caso contrario se obtendrá una nota de 0.
f. Si no se siguen las reglas del formato de email se obtendrá una nota de 0.
g. La nota de la documentación debe ser acorde a la completitud del proyecto.

14. La revisión de la documentación será realizada por parte del profesor, no durante la defensa del proyecto. El único requerimiento que se consultará durante la defensa del proyecto es el diagrama de clases, documentación interna y la documentación en el manejador de código.
15. Cada estudiante tendrá como máximo 15 minutos para exponer su trabajo al profesor y realizar la defensa de éste, es responsabilidad de los estudiantes mostrar todo el trabajo realizado, por lo cual se recomienda tener todo listo antes de ingresar a la defensa.
16. Cada excepción o error que salga durante la ejecución del proyecto y que se considere debió haber sido contemplada durante el desarrollo del proyecto, se castigará con 2 puntos de la nota final del proyecto.
17. Cada estudiante es responsable de llevar los equipos requeridos para la revisión, si no cuentan con estos deberán avisar al menos 2 días antes de la revisión a el profesor para coordinar el préstamo de estos.
18. Durante la revisión únicamente podrán participar el estudiante, asistentes, otros profesores y el coordinador del área.