Proyecto I – C!

Instituto Tecnológico de Costa Rica
Área de Ingeniería en Computadores
Algoritmos y Estructuras de Datos II (CE 2103)
Segundo Semestre 2017
Valor 25%

Objetivo General

- ➔ Diseñar e implementar clases que encapsulen el uso de punteros en C++

Objetivos Específicos

- ➔ Aplicar conceptos de manejo de memoria.
- ➔ Investigar y desarrollar una aplicación en el lenguaje de programación C++
- ➔ Investigar acerca de programación orientada a objetos en C++.
- ➔ Aplicar patrones de diseño en la solución de un problema.

Descripción del Problema

El proyecto consiste en el diseño e implementación de un nuevo tipo de dato en C++ denominado MPointer. MPointer es una clase template (MPointer<T>) que encapsula un puntero. Su labor consiste en crear una biblioteca para MPointer que pueda ser utilizada por cualquier programador para el desarrollo de una aplicación en C++. Deberá investigar cómo sobrecargar operadores como, & y * para MPointer.

Para facilitar la comprensión del problema, el proyecto se realizará en dos iteraciones (una sola revisión, se revisan dos entregables).

Primera iteración: encapsulamiento de punteros de C++

MPointer<T> tiene al menos un atributo de tipo T*. El programador que utiliza la biblioteca, crea un nuevo MPointer de la siguiente manera:

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

En este caso, aparte de reservar la memoria para MPointer, 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 MPointer, copia el puntero y el id de MPointer dentro del GC (más sobre esto adelante).

```
MPointer<int> myPtr = MPointer<int>::New();
MPointer<int> myPtr2 = MPointer<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 MPointer, se verifica si el tipo es el mismo a lo interno de MPointer. 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. MPointer tiene un destructor que se encarga de liberar la memoria del pointer guardado internamente.

MPointer no requiere de ningún runtime especial. Sin embargo, antes de usar MPointer, el programador debe inicializar la clase singleton MPointerGC que se ejecuta como un thread cada n segundos. Cada vez que se llama el método New de MPointer, se guarda la dirección de memoria de la instancia de MPointer dentro de la clase MPointerGC.

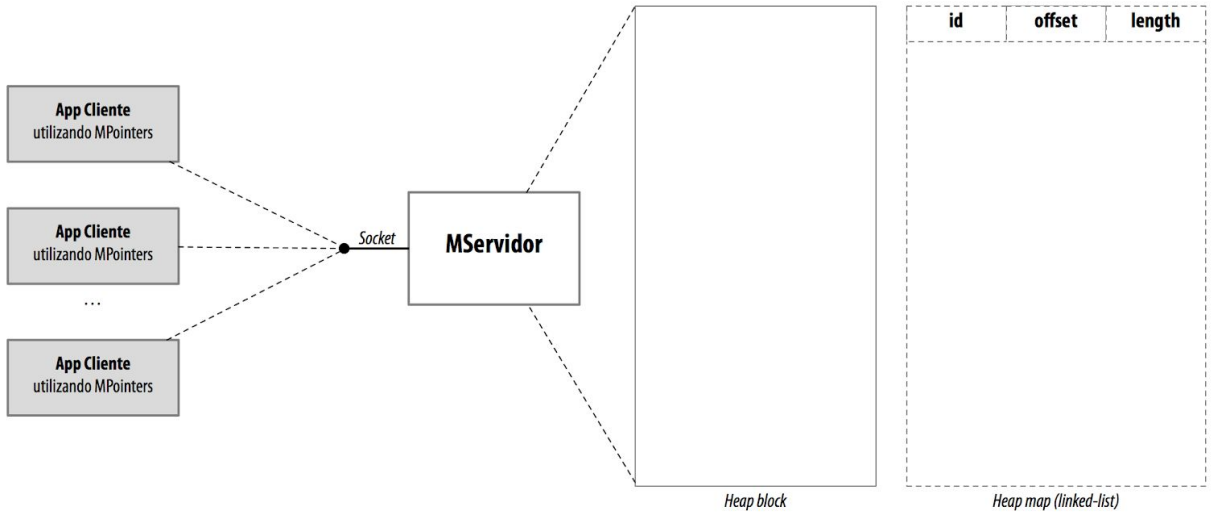
MPointerGC tiene una lista enlazada donde guarda direcciones de memoria de los MPointer conocidos.

MPointerGC le da a la instancia de MPointer, un Id autogenerado.

El destructor de MPointer llama a MPointerGC para indicar que la referencia se ha destruido. Una vez que el conteo de referencias de un MPointer llegue a cero, el MPointerGC lo libera, evitando memory leaks.

Para probar la funcionalidad de MPointer, deberá implementar QuickSort, BubbleSort e insertionSort utilizando listas doblemente enlazadas que utilizan MPointers internamente.

Segunda iteración: memoria distribuida
En esta iteración, se tendrá memoria distribuida, de la siguiente forma:



En esta modalidad, la biblioteca MPointers se usa de la siguiente forma:

1. El server se inicia, pasando como parámetro la cantidad de bytes que debe reservar. El server hace un único malloc y guarda la referencia al inicio de la memoria reservada.
2. El programador llama a MPointer_init pasando como parámetro el IP y puerto donde está el servidor
3. Cuando el programador llama al método New de MPointer, en vez de hacer malloc para el dato, va a enviar un mensaje al servidor solicitando memoria para almacenar el dato.

   ```
   MPointer<int> myPtr = MPointer<int>::New();
   ----> Envía mensaje al server en JSON indicando que reserve el espacio en el heap del server.

   <---- El server envía un id que se almacena internamente en MPointer. Este Id se utilizará posteriormente para leer o escribir memoria.
   ```

4. El servidor cuando recibe una solicitud de creación de memoria, utiliza el mapa de memoria para encontrar un espacio en el heap y crea una nueva entrada en el mapa de memoria
5. Cuando el programador asigna un valor al MPointer, se envia un mensaje al server indicando que salve el valor en memoria:

   ```
   *myPtr = 5.
   ----> Se envía un mensaje al server indicando que se actualice la memoria. MPointer almacena el id que se envia al server junto con el dato por guardar

   El server busca en el mapa de memoria el id indicado. Luego hace un desplazamiento en el heap y guarda el dato.
   ```

Documentación requerida

1. 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 grupos de 3 personas**.
4. Es obligatorio utilizar un GitHUB.
5. Es obligatorio integrar toda la solución.
6. El código tendrá un valor total de 75%, la documentación 20% y la defensa 5%.
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 del código NO podrá exceder en 35 puntos la nota de la documentación, por lo cual se recomienda documentar conforme se programa.
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++, 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.