



TD3: Algoritmos y Estructuras de Datos

Taller 2 (Memoria dinámica)

Introducción

En este taller van a trabajar a sobre un proyecto C++ configurado en el directorio `taller2-codigo`. Para empezar, asegúrense de poder abrir el directorio dentro del container correspondiente con VSCode, y de configurar el proyecto via CMake tal como indica el [apunte de configuración de entorno de desarrollo](#).

Notar que no todos los ejecutables del proyecto van a compilar inicialmente. A medida que avancen en los ejercicios irán extendiendo el código hasta poder compilar y ejecutar todos los tests provistos.

Para correr archivo de tests, elegir el ejecutable correspondiente (`test_parte<N>`) en la barra inferior, y compilarlo con el botón . Hecho esto van a poder ejecutar los tests desde el menú correspondiente () de la barra lateral, dentro del submenú *TestMate C++*.

Parte 1: Manejo básico de punteros

A modo de precalentamiento, la primera tarea tiene como objetivo practicar en el uso de las operaciones básicas requeridas para reservar y liberar memoria dinámica, y conocer los errores típicos que pueden surgir al trabajar con esas herramientas.

Abrir el archivo `test_parte1.cpp` y completar las áreas del código marcadas con comentarios `TODO` para hacer pasar los tests.

Recordatorio: `EXPECT_EQ(a, b)` hace pasar al test si `a==b`, y falla en caso contrario. Si todos los `EXPECT_EQ` pasan, entonces el test ejecuta satisfactoriamente.

Parte 2: Generalizando una lista enlazada

Partiendo de la implementación de lista enlazada sobre números provista en `lista.h`, extender el código para que la clase pueda manejar elementos de cualquier tipo genérico `T`.

Hecho esto deberán pasar los tests de `test_parte2.cpp`, sin necesidad de hacer ninguna modificación sobre dichos tests.

Parte 3: Operaciones de borrado

Extender `Lista<T>` para que soporte las siguientes operaciones:

1. `borrarUltimo`: dada una lista enlazada con al menos un elemento, borrar el elemento de la última posición. La memoria reservada para el elemento debe liberarse correspondientemente.
2. `borrarIesimo`: dada una lista enlazada con al menos “posicion” elementos, debe borrar el elemento de la posición pasada por parámetro. Acto seguido, modificar la implementación de `borrarUltimo` para que esté escrita en función de `borrarIesimo`.

Hecho esto deberán pasar los tests de `test_parte3.cpp` (¡y todos los anteriores!), sin necesidad de hacer ninguna modificación sobre dichos tests.

Parte 4: Agregando un Cursor

Se desea modificar la interfaz de la lista para que haya en todo momento un cursor apuntando a un elemento distinguido. El usuario podrá consultar cuál es el elemento apuntado por el cursor y avanzar el cursor para que apunte al elemento siguiente de la lista.

1. Implementar la operación **cursor**, que devuelve el elemento actualmente apuntado por el cursor. Detalles:
 - La operación está indefinida para listas vacías
 - Al agregarse un elemento a una lista vacía, el cursor automáticamente se enfoca en dicho elemento y queda fijo al agregar elementos adicionales hasta tanto no se avance manualmente.
 - Al borrar el elemento apuntado por el cursor se desea lo siguiente:
 1. De haber un elemento siguiente al actualmente apuntado, mover el cursor para que apunte al mismo
 2. Si no hay elemento siguiente, se intenta apuntar al cursor al elemento anterior
 3. Si el elemento borrado es el último de la lista, el cursor queda indefinido (pero deberá respetar el comportamiento especificado anteriormente al volver a agregar elementos)
2. Implementar la operación **avanzar**, que cambia el elemento al que apunta **cursor** a su siguiente. Si **cursor** apunta al último elemento, la operación no tiene ningún efecto.

A esta altura deberán pasar los tests de `test_parte4.cpp` (¡y todos los anteriores!), sin necesidad de hacer ninguna modificación sobre dichos tests.

Parte 5: Cursor con retroceso

Para permitir al usuario volver a enfocar elementos anteriores con el cursor, se pide:

1. Hacer que `Lista<T>` esté doblemente enlazada, de modo que cada nodo tenga también un puntero al elemento anterior. Adaptar el código de `Lista<T>` para actualizar los enlaces entre nodos de forma coherente al agregar y quitar elementos.
2. Implementar la operación **retroceder**, que cambia el elemento al que apunta el cursor a su anterior. Si **cursor** apunta al primer elemento, la operación no tiene ningún efecto.

A esta altura deberán pasar los tests de `test_parte5.cpp` (¡y todos los anteriores!), sin necesidad de hacer ninguna modificación sobre dichos tests.

Referencia: Manejo de memoria dinámica

```
// -----  
// Usamos "new" para reservar la memoria necesaria para almacenar  
// un valor de cualquier tipo.  
new tipo_de_dato(valor_a_asignar);  
  
// Usamos la siguiente sintaxis cuando el tipo de dato es un struct .  
// Los valores deben ir en orden. Ver nodo en ListaEnlazada para un ejemplo.  
new tipo_de_dato {valor_variable_1, valor_variable_2, ... };  
  
// -----  
// Usamos "delete" para liberar el espacio de memoria referenciado por un puntero.  
// Es una buena práctica asignar el puntero a nullptr luego de liberar la memoria.  
delete variable_puntero;  
  
// -----  
// C++ provee una sintaxis más compacta al operar sobre punteros a structs e  
// instancias de clases. Las dos siguientes expresiones son equivalentes:  
(*puntero_a_instancia).metodo()  
puntero_a_instancia->metodo()  
  
// La sintaxis simplificada también aplica al acceso de  
// variables internas de objetos y structs  
(*puntero_a_instancia).variable_interna  
puntero_a_instancia->variable_interna
```

Referencia: Templates

```
// Utilizamos el keyword `template` para construir clases paramétricas,  
// declarando la(s) variable(s) de tipo correspondiente.  
//  
// Recordar que la definición de un template debe estar contenida en un archivo .h.  
// No es posible separar el código en definición (.h) e implementación (.cpp) como  
// solemos hacer para clases concretas.  
template <typename T>  
class NombreDeLaClase {  
    // ... aquí va el cuerpo de la clase  
}  
  
// Luego, en el código podemos instanciarla de la siguiente forma:  
void mi_funcion() {  
    // ... proveyendo los argumentos del constructor asociado  
    NombreDeLaClase<tipo_de_dato> mi_instancia(...)  
}  
  
// La implementación de los métodos de tipos genéricos también  
// se definen como templates:  
template <typename T>  
tipo_de_dato NombreDeLaClase<T>::nombre_del_metodo(T nombre_param) {  
    // ... aquí va el cuerpo del método  
}
```