Informe del primer Seminario de LP

Nota: Todas las secciones sin texto son ejercicios de implementación, las respuestas se pueden encontrar en los archivos anexos al informe.

Nota: Se provee de un documento *pdf* anexo con consideraciones extras en inglés acerca de algunos de los temas tratados en el seminario.

1

1.1 Introducir lo que es un template en C++ enfocado a la genericidad y cómo funciona de manera abreviada

Un template es una manera especial de escribir funciones y clases para que estas puedan ser usadas con cualquier tipo de dato y esta lo que hace es sustituir código en tiempo de compilación.

2

2.1 ¿Qué significan por valor, por puntero y por referencia en C++? ¿Cómo funciona esto en memoria?

1. Por valor: Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a "objetos" locales de la función. Esto se hace moviendo todo el "objeto" para el stack.

2. Por puntero: Sirven para indicar la posición de un "objeto" en la memoria. Esta en memoria se maneja semejante al método por valor, se pasa todo este número de un lugar a otro en los llamados a funciones.

3. Por referencia: Las referencias sirven para definir "alias" o nombres alternativos para un mismo objeto. Esta en memoria se maneja de una manera especial en la que el compilador genera código que hace que el valor en la pila referencie directamente al "objeto" en la memoria.

2.2 ¿Cuál es la filosofía en el uso de la memoria defendida por C++?

En C++ el programador maneja la memoria casi por completo. El compilador solo se encarga de liberar la memoria que fue reservada en un contexto pero

esta no fue creada por el operador new. Todos los recursos reservados en el *heap* deben ser liberados usando los operadores delete y delete[].

La filosofía de manejo de memoria en C++ es **determinista**, en cualquier punto del programa se pueden conocer todos los objetos actualmente en la memoria dinámica o estática, algo que en lenguajes como C# o Java no es posible. C++ no cuenta con *Garbage Collector*.

3 Definir constructores básicos de C++ y el operador =

3.1 ¿Qué hace cada uno de ellos? ¿Cuándo se llaman?

Hay varios constructores implementados en la clase node, cada uno de ellos es una sobrecarga para distintos tipos de parametros a la hora de crearlos, también está sobrecargado el operador = para que se pueda crear un nodo directamente asignando un valor genérico. El compilador sabe cuando llamar a cada uno ya que escoge el más restrictivo dado los parametros recibidos.

3.2 Explicar la inicialización de campos

La inicialización de campos es establececr los valores que tendrán las distintas propiedades de la estructura en el momento de crearla.

3.3 ¿Se puede hacer list-initialization al estilo C#?

Inicializar una lista mediante llaves, en la forma vector <int> a = {1,2,3,4 }, esta disponible solo a partir de C++11.

3.4 ¿Cómo funciona el paso de parámetros cuando se llama a una función?

Parámetro formal: una variable y su tipo tal como aparecen en el prototipo de la función o método.
Paso de parámetros en funciones:

1. Paso por valor: Cuando se pasan parámetros por valor, el compilador realiza una copia de ellos en una zona de memoria separada, por lo que los cambios realizados en el parámetro no se transmiten de nuevo a la persona que llama. Cualquier modificación a la variable de parámetro dentro de la función o método llamado afecta solo a la ubicación de almacenamiento en donde se copia el parámetro de la función y no se reflejará en el parámetro real en el entorno de llamada.

2. Paso por referencia: La llamada por método de referencia para pasar argumentos a una función copia la referencia de un argumento(alias) en el parámetro formal. Dentro de la función, la referencia se utiliza para

acceder al argumento real utilizado en la llamada. Esto significa que los cambios realizados en el parámetro afectan el argumento pasado.

3. Paso por puntero: El método de llamada por puntero para pasar argumentos a una función copia la dirección de un argumento en el parámetro formal. Dentro de la función, la dirección se utiliza para acceder al argumento real utilizado en la llamada. Esto significa que los cambios realizados en el parámetro afectan el argumento pasado.

3.5 ¿Cuándo se deben utilizar parámetros por valor, por puntero o por referencia?

1. Paso por valor: Se usa para mantener un código más seguro, en el que no se quisiera que la función modifique el valor pasado como parámetro. Si se tienen dos funciones f(int a) y g(int a), las cuales para su ejecución modifican el parámetro de entrada, y se tiene una variable a y se desea obtener f(a) y g(a), es conveniente llamar a f y a g por paso por valor.

2. Paso por referencia o puntero: Se usa para obtener un código más rápido, eficiente, y que ahorra memoria. La desventaja de esto es que el programa puede ser difícil de leer, además de que se podrían producir errores lógicos en tiempo de ejecución.

3.6 Constructores con un solo argumento

Al pasar parámetros a una función, si los tipos no coinciden, el compilador puede realizar una conversión implícita. Esto significa que el compilador puede hacer uso de un constructor que tome un solo parámetro y usarlo para convertir el tipo de parámetro de entrada al tipo correcto que espera la función. Un ejemplo es el código a continuación

```
class B{
     int a;
     int b;
     public:
          B(int x){
               a = x;
               b = x *10;
          }
     int get(){return b;}

};

void f(B b){
     cout<<b.get()<<endl;
}
```

Si llamamos a la función f en el entero 10 por ejemplo (f(10)), el compilador convertirá implícitamente el entero 10 en un objeto de tipo B, llamando al constructor de un solo parámetro q recibe un entero. Por lo que la función imprimirá el entero 100.

3.7 Constructores explicit

En caso de que no se desee que el compilador convierta implícitamente debido a que se pueden producir conversiones no previstas se puede usar la palabra reservada *explicit*. Si un constructor de conversión posee esta palabra reservada, el compilador no podrá convertir tipos a menos que el programador explícitamente lo pida. Siguiendo con el ejemplo anterior, en este caso habría q llamar a la función f de esta forma: f(B(100)).

4

4.1

5

5.1 ¿Qué es un destructor?

El destructor es el que se encarga de liberar toda la memoria que utilizó el "objeto" asociado.

5.2 ¿Cuándo se debe definir el destructor como virtual?

Las funciones virtuales se utilizan para aplicar polimorfismo entre las clases bases abstractas y las clases que se derivan de ellas. Así, si una clase posee una función virtual pura, ésta pasa a ser una clase abstracta, por lo q las clases concretas q deriven de ella deben implementar dicha función.
Los destructores virtuales son útiles cuando queremos eliminar una instancia de una clase derivada que se está apuntando a ella mediante un puntero de clase base. En caso de no ser virtual el destructor de la clase base, se obtiene *undefined behavior*, puesto que se podría llamar al destructor de la clase base, haciendo que la clase derivada no se elimine bien. Luego, si definimos el destructor de la clase base como virtual, se llamará al destructor de la clase derivada y luego al destructor de la clase base (note que la clase base no necesariamente debe ser abstracta, para ser abstracta necesita una función virtual pura).

5.3 Explicar los operadores delete y delete[]

Para reservar memoria dinámicamente se usa el operador *new*. Para reservar memoria mediante un puntero se puede usar *int* ptr = (int*)malloc(sizeof(int))* . Pero también se puede reservar un array de elementos mediante *int* array = new int[10]*. Ahora, los operadores delete y delete[] son la contraparte de estas

dos formas de reservar memoria mediante el operador new. O sea, delete elimina memoria reservada mediante objetos que no son arrays, delete[] destruye arrays creados por el operador new.

6

6.1 Funciones inline v.s Macros de C

Las funciones inline ayudan a reducir la sobrecarga por llamadas de función, en especial las funciones con un cuerpo pequeño. El calificador inline colocado en la definición de la función antes del tipo de regreso le pide al compilador q genere una copia del código de la función cada vez q vea una llamada a esa función, con el fin de evitar una llamada a función en tiempo de ejecución. El compilador puede ignorar esta petición del programador, y lo hará en los casos en que el cuerpo de la función sea grande.
Ventajas de funciones inline sobre macros de C:

1. Las funciones inline, al ser funciones, aceptan verificación de tipos, algo que las macros de C no hacen.
2. Las funciones inline eliminan los efectos colaterales inesperados, asociados con un uso inapropiado de las macros.
3. Las funciones inline pueden ser depuradas.

6.2 ¿Cuándo usar funciones inline?

Cuando el cuerpo de la función sea pequeño.

6.3 ¿Cómo se comportan las variables por valor, punteros y las referencias como retorno de una función?

Retornar valores por valor, referencia o puntero se comporta muy parecido a pasar parámetros por valor, referencia o puntero. La principal diferencia es que el flujo de los datos es en reversa, y además hay que considerar que al retornar la función, las variables locales pierden su scope. Teniendo esto en cuenta analicemos cada caso.

1. Retorno por valor: Retonar por valor es la forma más segura y simple de retorno de una función. Mediante este retorno no hay que preocuparse por el scope de las variables debido a que se genera una copia de la variable que se desea retornar. Este tipo de retorno se usa cuando la variable que se desea retornar fue declarada en el cuerpo de la función, o es un parámetro por valor que se desea retornar.
2. Retorno por puntero: La función retorna la dirección de una variable definida en su cuerpo. Debido a que solo se copia la dirección de la variable

a retornar, este modo de retorno es rápido. Sin embargo, el retorno por puntero tiene una gran debilidad. Si queremos devolver la dirección de una variable definida en el cuerpo de la función, como el scope de la variable solo llega a la función, al terminar ésta, la zona de memoria de la variable queda *non-allocated*. Por lo que obtendríamos un puntero a una zona de memoria *non-allocated*. Al anterior puntero se le llama puntero colgante, el cual al usarse se obtendrá *undefined behavior*. Lo que si se puede retornar es la dirección de la memoria reservada mediante la palabra reservada *new*, debido a que el scope de la memoria reservada por el programador no se limita al cuerpo de la función. En resumen, este modo de retorno se usa cuando se quiere devolver memoria reservada por el programador, o si se quiere devolver un parámetro que se pasó por puntero.

3. Retorno por referencia: Devuelve una referencia a una variable en el cuerpo de la función. Comparte la misma debilidad que con el anterior tipo de retorno. Es usado cuando se quiere devolver una referencia a un elemento de un array, algo útil cuando se implementan clases que poseen estructuras parecidas a arrays. También se usa si se quiere devolver estructuras o instancias de clases grandes, las cuales no serán destruidas al final de la función.

6.4 Explicar las funciones const

Una función con el modificador const asegura que no se modificará el objeto al cual pertence la función. Una función const no pordrá llamar a funciones non-const. Cualquier intento de una función const de cambiar una variable del objeto o llamar a una función que no es const resultará en error de compilación. Los constructores no pueden ser marcados como const, debido a que el constructor debe ser capaz de inicializar los valores del objeto, por lo que el lenguaje no permite constructores const. Es una buena práctica nombrar a una función que no modifique el objeto, const, para que así pueda ser llamada por otras funciones const.

6.5 ¿Cómo se capturan y lanzan las excepciones?

Las excepciones se lanzan usando *throw* y se capturan usando los metódos de try y catch() en donde si se lanza una excepción en el try, esta es recogida en los parametros del catch.

7

7.1

7.2 Limitaciones del operador []

1. La sobrecarga del operador [] no puede ser estática. Esto es para asegurar que siempre se tenga un left value.

2. El operador [] debe ser definido como función miembro de una clase.

7.3

7.4

El tipo de retorno debe ser por valor ya que este solo guarda los punteros a los nodos first y last, y el tamaño de la linkedlist. Los nodos fueron creados en el heap por lo que nunca se borraron al terminar el scope en este constructor.

8

8.1

8.2

8.3 ¿Se puede crear un Function<R, T...> con un número variable de Ts?

Se puede hacer a partir de C++11 mediante el uso de *variadic template*, lo cual permite definir templates con un número arbitrario de parámtros. En el caso de C++ antes de C++11 no es posible. A continuación una posible solución sin un número variable de parámetros.

```
template<typename T, typename R>
struct Function {
    typedef R(*Value)(T);
};

template<typename T, typename R>
linked_list<R> Map(typename Function<T, R>::Value mapper,
                   linked_list<T> ll){
    linked_list<R>* result = new linked_list<R>();

    for (int i = 0; i < ll.Length(); i++)
        result->Add_Last(mapper(ll.At(i)));

    return result;
}
```

9

9.1 Uso de la palabra reservada friend

Una función amigo de una clase se define por fuera del alcance de dicha clase, pero aún así tiene el derecho de acceso a los miembros private y protected. Se puede declarar una función o toda una clase como un *friend* de otra clase. Para declarar una función como un friend de una clase, en la definición de clase preceda el prototipo de función con la palabra reservada *friend*. Para declarar la clase A como amigo de la clase B, se debe hacer *friend A;* como un miembro de la clase B. Para que la clase B sea amigo de la clase A, la clase A debe declarar que la clase B es su amigo. La relación *friend* no es simétrica ni transitiva.