Lenguajes de Programación

Seminario#1
C++11, C++14.

Equipo 2:
Marcos Manuel Tirador del Riego
Laura Victoria Riera Pérez
Leandro Rodríguez Llosa

Grupo: C-311

Índice

1. Definición de las clases genéricas linked_list y node

Listing 1: Sample Code Listing C++

```
#include <iostream>

int main()
{
  std::cout << "Hello World";
  return 0;
}
```

Una lista enlazada es una estructura de datos compuesta de nodos, donde cada nodo contiene alguna información y un puntero a otro nodo de la lista. Si un nodo tiene sólo un enlace con su sucesor en esta secuencia, la lista se denomina lista enlazada simple.

2. Definición de miembros de datos necesarios de ambas clases

2.1. Nuevos elementos introducidos a partir de C++11 que permiten un manejo más ïnteligente"de la memoria

Se introduce el principio "la adquisición de recursos es inicialización" o *RAII*, con lo cual se reserva o libera espacio en memoria.

El C++ moderno evita usar la memoria del heap tanto como sea posible declarando objetos en la pila. Cuando un recurso es demasiado grande para la pila, debe ser propiedad de un objeto. A medida que el objeto se inicializa, adquiere el recurso que posee. El objeto es entonces responsable de liberar el recurso en su destructor. El propio objeto propietario se declara en la pila.

Cuando un objeto de la pila propietario de recursos queda fuera del alcance, su destructor se invoca automáticamente. De esta manera, la recolección de basura en C++ está estrechamente relacionada con la vida útil del objeto y es determinista, a diferencia de los recolectores de basura utilizados por otros lenguajes. Un recurso siempre se libera en un punto conocido del programa, que puede controlar. Solo los destructores deterministas como los de C++ pueden manejar los recursos de memoria y los que no son de memoria por igual.

Los elementos introducidos con este fin son los punteros inteligentes, quienes apuntan a objetos y, cuando el puntero sale del alcance (scope), el objeto se destruye. Esto los hace inteligentes en el sentido de que no tenemos que preocuparnos por la liberación manual de la memoria asignada. Los punteros inteligentes hacen todo el trabajo pesado por nosotros.

2.2. Inicialización

En C++11 hay cuatro tipos de punteros inteligentes:

- `std::auto_ptr`: es un remanente en desuso de C++98. Fue un intento de estandarizar lo que más tarde se convirtió en std::unique_ptr de C++11. Hacer el trabajo bien requiere move semantics, pero C++98 no las tenía. Como solución alternativa, std::auto_ptr cooptó sus operaciones de copia para movimientos. Esto condujo a un código sorprendente (copiar un std::auto_ptr lo establece en nulo) y restricciones de uso frustrantes (por ejemplo, no es posible almacenar std::auto_ptrs en contenedores).

- `std::unique_ptr`: encarna la semántica de propiedad exclusiva.

  Un estándar no nulo `std::unique_ptr` siempre posee a lo que apunta.

  Mover un `std::unique_ptr` transfiere la propiedad del objeto desde el puntero de origen hasta el puntero de destino (el puntero de origen es establecido en nulo). No está permitido copiar un `std::unique_ptr`, porque si se pudiera, terminaría con dos `std::unique_ptr` para el mismo recurso, cada uno pensando que lo poseía (y por lo tanto debería destruirlo). `std::unique_ptr` es, por lo tanto, un tipo de solo movimiento.

  Tras la destrucción, un `std::unique_ptr` no nulo destruye su recurso. Por defecto, la destrucción de recursos es realizada aplicando `delete` al raw pointer dentro de `std::unique_ptr`, pero se puede especificar su forma de destrucción.

  Es razonable asumir que, por defecto, los `std::unique_ptr` poseen el mismo tamaño que los raw pointers, y para la mayoría de las operaciones (incluida la desreferenciación), ejecutan exactamente las mismas instrucciones. Esto significa que pueden ser usados incluso en situaciones donde la memoria y los ciclos son apretados.

- `std::shared_ptr`: Un objeto al que se accede a través de `std::shared_ptr` tiene su vida útil administrada por esos punteros a través de propiedad compartida.

  Ningún `std::shared_ptr` específico posee al objeto. En cambio, todo `std::shared_ptr` apuntando a este colabora para asegurar su destrucción en el punto donde ya no se necesite. Cuando el último `std::shared_ptr` que apunta a un objeto deja de apuntar allí (por ejemplo, porque el `std::shared_ptr` se destruye o apunta a un objeto diferente), este destruye el objeto al que apunta.

  Como con la recolección de basura, los programadores no necesitan preocuparse por administrar la vida tiempo de los objetos referenciados, pero como con los destructores, el momento de la destrucción de los objetos es determinista.

  Un `std::shared_ptr` puede decir si es el último que apunta a un recurso consultando el conteo de referencia del mismo (valor asociado con el objeto que mantiene seguimiento de cuántos `std::shared_ptr` apuntan a él). Los constructores de `std::shared_ptr` incrementan este conteo, los destructores lo disminuyen y los operadores de asignación de copia hacen ambas cosas. Si un `std::shared_ptr` ve un conteo de referencia con valor cero después de realizar una disminución, no hay ningún otro `std::shared_ptr` apuntando al recurso, por lo que lo destruye.

  En comparación con `std::unique_ptr`, los objetos `std::shared_ptr` suelen el doble de grande, generan gastos generales para los bloques de control y requieren manipulaciones de conteo de referencia atómicas.

  La destrucción de recursos predeterminada se realiza mediante eliminación, pero se admiten destructores personalizados.

- `std::weak_ptr`: actúa como un `std::shared_ptr`, pero no participa en la propiedad compartida del recurso apuntado, y, por lo tanto, no afecta el conteo de referencias del mismo. En realidad hay un segundo conteo de referencia en el bloque de control, y es este el que `std::weak_ptr` manipula.

  Este tipo de puntero inteligente tiene en cuenta un problema desconocido para `std::shared_ptr`: la posibilidad de que a lo que apunta haya sido destruido. `std::weak_ptr` soluciona este problema rastreando cuando cuelga (*dangles*), es decir, cuándo el objeto al que se supone que apunta ya no existe.

A menudo lo que desea es comprobar si un `std::weak_ptr` ha caducado y, si no, acceder al objeto al que apunta.

Desde una perspectiva de eficiencia, los `std::weak_ptr` son iguales que los `std::shared_ptr`. Los objetos `std::weak_ptr` tienen el mismo tamaño que `std::shared_ptr` objetos, hacen uso de los mismos bloques de control que `std::shared_ptr` y operaciones como construcción, destrucción y la asignación implica manipulaciones de conteo de referencias atómicas.

Los posibles casos de uso de `std::weak_ptr` incluyen el almacenamiento en caché, las listas de observadores y la prevención de ciclos `std::shared_ptr`

Todos están diseñados para ayudar a manejar el tiempo de vida de objetos asignados dinámicamente, es decir, para evitar fugas de recursos al garantizar que tales objetos se destruyen de la manera apropiada en el momento apropiado (incluyendo en caso de excepciones).

2.3. Filosofía en el uso de la memoria defendida por C++

2.4. Simplificación de nombres de tipos mediante el uso de alias

3. Definición de los constructores clásicos de C++(C++0x) , el constructor move y las sobrecargas del operador =

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

3.2. ¿Qué es un lvalue y un rvalue ?

3.3. std::move

4. Definición de un constructor que permita hacer list-initialization lo más parecido a C# posible

4.1. Compare la utilización del {} v.s ()

5. Definición de un constructor que reciba un vector $< T >$

5.1. Usar for_each con expresiones lambda

6. Definición del destructor de la clase

6.1. ¿Hace falta?

6.2. ¿Para qué casos haría falta un raw pointer?

Un puntero es un tipo de variable. Almacena la dirección de un objeto en la memoria y se utiliza para acceder a ese objeto. Un raw pointer es un puntero cuya vida útil no está controlada por un objeto encapsulador, como un puntero inteligente. A un puntero sin procesar se le puede asignar la dirección de otra variable que no sea un puntero, o se le puede asignar un valor de nullptr. Un puntero al que no se le ha asignado un valor contiene datos aleatorios.

También se puede desreferenciar un puntero para recuperar el valor del objeto al que apunta. El operador de acceso a miembros proporciona acceso a los miembros de un objeto.

En C++ moderno, los raw pointers solo se usan en pequeños bloques de código de alcance limitado, bucles o funciones auxiliares donde el rendimiento es crítico y no hay posibilidad de confusión sobre la propiedad.

7. Definición de las funciones length , Add_Last , Remove_Last , At , Remove_Ate

7.1. Noexcept

7.2. Inferencia de tipo en C++ (auto, decltype decltype(auto)). Explicar todos, pero no obligatoriamente usarlos.

8. Crear un puntero a función Function¡R, T...¿que devuelve un valor de tipo R y recibe un número variable de parámetros de tipo T .

8.1. Definir una función genérica Map a linked_list en T y R , que recibe un puntero a función que transforma un elemento T en uno R; de manera que Map devuelve una instancia de linked_list<R> resultado de aplicar a todos los elementos T de la lista original la función de transformación.

8.2. Crear punteros a funciones usando alias

8.3. Crear un puntero a función Function que permita cualquier cantidad de parámetros de cualquier tipo.