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

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 antecesor y su sucesor en esta secuencia, la lista se denomina lista doblemente enlazada.

```
template <typename T>
class node
{
private:
    node *next{nullptr};
    node *previous{nullptr};

public:

    T value;
    node() : value(0){}
    node(T val) : value(val){}

    node<T> *Next();
    node<T> *Previous();
    bool HasNext();
    bool HasPrevious();
    void SetNext(node<T> &son);
    void SetPrevious(node<T> &parent);
    void DeleteNext();
    void DeletePrevious();
    void CreateNext(T value);
    void CreatePrevious(T value);

};

template <typename T>
class linked_list
{
    using node_T = node<T>; //alias

public:
    int size;
    linked_list() : size(0) { head = nullptr, tail = nullptr; }; // ctor

    void Add_Last(T val);
    void Remove_Front();
    void Remove_Last();
    void Remove_At(int pos);
    node_T *At(int i);
    T operator[](int i);

private:
    node_T *head;
    node_T *tail;
};
```

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 "inteligente"de la memoria

Se introduce el principio "la adquisición de recursos es inicialización" o por sus siglas en inglés *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_ptr` 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.

  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 ser el doble de grandes, 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`, 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++

En C++, podemos asignar de manera eficiente la memoria en tiempo de ejecución y desasignarla cuando no se requiera. Con esta función, obtenemos la flexibilidad de asignación y desasignación de memoria según los requisitos. Con este fin se introducen los punteros inteligentes, ya mencionados

anteriormente, y contenedores (estructuras de datos quienes se ocupan de la gestión del espacio de almacenamiento necesario).

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.

Además como se mencionó en [2.1] en el C++ moderno solo se reserva memoria en el heap cuando es absolutamente necesario y se necesita asignar un gran bloque de memoria. Al contrario si se trabaja con variables relativamente pequeñas que solo se requieren mientras una función está ejecutándose es mejor utilizar la pila debido a que provee un acceso más fácil y rápido.

Siguiendo la misma filosofía para ganar en espacio y eficiencia, se comportan otros componentes del lenguaje, por ejemplo, siempre que sea posible pasar por referencia en lugar de por valor, utilizar semánticas de movimiento (move semantics) en lugar de copia, etc.

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

Una declaración de alias se utiliza para declarar un nombre que se usará como sinónimo de un tipo declarado previamente, de igual forma que typedef. La diferencia es que los alias también admiten la creación de plantillas, las cuales son una manera especial de escribir funciones y clases para que estas puedan ser usadas con cualquier tipo de dato (similar a la sobrecarga en el caso de las funciones, pero evitando el trabajo de escribir cada versión de la función). En este caso se llaman plantillas de alias y pueden resultar útiles para los asignadores personalizados.

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

```
1
2
3  template <typename T>
4  class linked_list
5  {
6      using node_T = node<T>;
7
8  public:
9      int size;
10     linked_list() : size(0) { head = nullptr, tail = nullptr; }; // ctor
11
12     #pragma constructors
13
14
15         linked_list(const linked_list &a) //copy ctor
16         {
17             this->operator=(a);
18         }
19         linked_list & operator=(const linked_list &a) //copy assignment ctor
20         {
21             size = 0;
22             head = tail = nullptr;
23             if(a.size == 0) {return *this;}
24             node_T* nod = a.head;
25             this->Add_Last(nod->value);
```

```
26              while(nod->HasNext())
27              {
28                  nod =  nod->Next();
29                  this->Add_Last(nod->value);
30              }
31              return *this;
32          }
33          linked_list (linked_list && a) //move ctor
34          {
35              head = a.head;
36              tail = a.tail;
37              size = a.size;
38              a.head = a.tail = nullptr;
39              a.size = 0;
40          }
41          linked_list & operator=(linked_list && a) //move assignment ctor
42          {
43              head = a.head;
44              tail = a.tail;
45              size = a.size;
46              a.head = a.tail = nullptr;
47              a.size = 0;
48
49              return *this;
50          }
51
52
53
54      #pragma endregion
55
56  private:
57      node_T *head;
58      node_T *tail;
59  };
```

3.1. ¿Qué es un lvalue y un rvalue?

Cada expresión de C++ tiene un tipo y pertenece a una categoría de valor. Las categorías de valor son la base de las reglas que los compiladores deben seguir al crear, copiar y mover objetos temporales durante la evaluación de expresiones.

Un *lvalue* es algo que apunta a una ubicación de memoria específica, produce una referencia a un objeto, como un nombre de variable, una referencia de subíndice de matriz, un puntero sin referencia o una llamada de función que devuelve una referencia. Un lvalue siempre tiene una región de almacenamiento definida, por lo que puede tomar su dirección.

Un *rvalue* es algo cuya dirección no se puede obtener desreferenciandolos, es decir, no apuntan a ninguna parte. Los ejemplos de rvalues incluyen literales, los resultados de la mayoría de los operadores y llamadas a funciones que no devuelven referencias. Un rvalue no tiene necesariamente ningún almacenamiento asociado. En general, los rvalues son temporales y de corta duración.

En pocas palabras, un lvalue es una referencia a un objeto y un rvalue es un valor.

3.2. Constructores

Un constructor es una función que tiene el mismo nombre que la clase encargado de inicializar un objeto de esta. Construye un objeto y puede establecer valores para los miembros de datos. Este se invoca cuando tiene lugar la inicialización del objeto, no puede ser llamado directamente.

- Constructores clásicos:

  El constructor predeterminado es aquel que se puede llamar sin argumentos (no tiene parámetros o estos están predeterminados). Si este no está definido explícitamente en el código, el compilador genera uno por defecto.

  Los constructores pueden tener parámetros arbitrarios y ser declarados explícitamente, estos son constructores proporcionados por el usuario (programador).

- Constructor copy:

  Se utiliza para crear un nuevo objeto a partir de otro existente. Tiene como parámetro de entrada una referencia a otro objeto de la misma clase, de forma tal que las variables del objeto que se está creando se inicializan con los valores del objeto a copiar. Si no se define, el sistema proporciona uno, el cual realiza una copia bit a bit entre los objetos.

- Constructor move:

  El constructor move permite que los recursos que pertenecen a un objeto rvalue se muevan a un lvalue sin hacer copias, lo cual lo hace eficiente en términos de velocidad de ejecución, y acepta como argumento una referencia rvalue. Este "roba"los recursos contenidos en el argumento, (por ejemplo, punteros a objetos asignados dinámicamente, descriptores de archivos, conectores TCP, flujos de E/S, subprocesos en ejecución, etc) y dejan el argumento (objeto desde el que se movieron los datos) en algún estado válido pero no especificado. Si un usuario no declara un constructor move, el compilador proporciona uno predeterminado.

3.3. Sobrecargas

C++ permite especificar más de una función con el mismo nombre en el mismo ámbito. Estas funciones se denominan funciones sobrecargadas o sobrecargas. Las funciones sobrecargadas permiten proporcionar una semántica diferente para una función, dependiendo de los tipos y el número de argumentos.

Podemos sobrecargar los siguientes operadores: Operadores aritméticos, operadores binarios, operadores booleanos, operadores unarios, operadores de comparación, operadores compuestos, operadores de función y subíndice.

Cada operador lleva su firma y un conjunto de reglas cuando se sobrecarga por clases.

Sobrecarga del operador = :

Un operador de asignación se implementa mediante una función miembro no estática con exactamente un parámetro.

Este puede ser sobrecargado y se utiliza para copiar valores de un objeto a otro objeto ya existente, en este caso se conoce como operador de asignación de copia (copy). Por otro lado si declaramos un objeto y luego intentamos asignar una referencia rvalue a él (el objeto que aparece en el lado derecho de una expresión de asignación es un rvalue) se invoca el operador de asignación de movimiento (move) que utiliza move semantics.

3.4. La función std::move

`std::move` se usa para permitir la transferencia eficiente de recursos de t a otro objeto. Al pasar un objeto a esta función se obtiene un rvalue que hace referencia a él.

Los nombres de las variables de referencia de rvalue son lvalues y deben convertirse a valores para vincularse a las sobrecargas de funciones que aceptan parámetros de referencia de rvalue, razón por la cual los constructores move y los operadores de asignación move suelen usar `std::move`.

Esta es una función auxiliar para forzar la semántica de movimiento en los valores, incluso si tienen un nombre pues el uso directo del valor devuelto hace que arg se considere un rvalue. Permite transferir la propiedad de los activos y las propiedades de un objeto directamente cuando el argumento es un rvalue sin tener que hacer copias costosas.

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

```
linked_list(initializer_list<T> lst)
        {
            size = 0;
            head = tail = nullptr;
            for(auto it = lst.begin();it!=lst.end();++it)
            {
                this->add_element(*it);
            }
        }
```

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

Los valores de inicialización se pueden especificar con paréntesis, un signo igual o llaves.

Para abordar la confusión de múltiples sintáxis de inicialización, así como el hecho de que no cubren todos los escenarios de inicialización, C++11 introduce una inicialización uniforme o universal, una sintáxis de inicialización única que puede usarse en cualquier lugar y expresar cualquier cosa (asignación, inicialización, inicialización directa en constructores, etc), mediante términos entre llaves.

La inicialización con llaves permite expresar lo que antes era inexpresable. Funciona en la mayoría de los contextos, sin construcción de copia implícita (a diferencia de "="). Se pueden usar para especificar valores de inicialización predeterminados para datos no estáticos. Esta capacidad, nueva en C++ 11, se comparte con la sincronización de inicialización "=", pero no con paréntesis.

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

5.1. Usar for_each con expresiones lambda

```
linked_list(vector<T> & v)
        {
            size = 0;
            head = tail = nullptr;
            for_each(v.begin(),v.end(),[&](T x){this->Add_Last(x);});
        }
```

6. Definición del destructor de la clase

Como se vió anteriormente, un constructor es una función miembro que se invoca cuando el objeto se inicializa. De manera similar, un destructor es una función miembro que se invoca cuando un objeto es destruido. No toma parámetros, y hay uno por clase. El nombre del destructor es el símbolo $\sim$ seguido del nombre de la clase. Si el usuario no define un destructor, el compilador proporciona uno por defecto.

```
linked_list()  //destructor
        {
            if(size == 0){return;}
            node_T* nod = head;
            while(nod->HasNext())
            {
                node_T* nod2 = nod->Next();
                delete nod;
                nod = nod2;
            }
            delete nod;
        }
```

6.1. ¿Hace falta?

Los destructores son necesarios pues se llaman cuando un objeto sale del alcance o cuando un puntero a un se elimina el objeto, liberando la memoria dinámica utilizada por dicho objeto o liberando recursos empleados.

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 crudo 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_At*

```
#pragma implementation_of_list_members

template <typename T>
void linked_list<T>::add_element(T val)
{
    if (!size)
```

```
7     {
8         head = tail = new node<T>(val);
9         size = 1;
10        return;
11    }
12    tail->CreateNext(val);
13    // *(tail -> next) = nod;
14    tail = tail->Next();
15    size++;
16    if (size == 2)
17    {
18        head->SetNext(*tail);
19    }
20 }
21 template <typename T>
22 void linked_list<T>::pop_front()
23 {
24    if (!size)
25    {
26        throw range_error("");
27    }
28    delete_at(0);
29 }
30 template <typename T>
31 void linked_list<T>::pop_element()
32 {
33    if (!size)
34    {
35        throw range_error("");
36    }
37    delete_at(size - 1);
38 }
39 template <typename T>
40 void linked_list<T>::delete_at(int pos)
41 {
42    if (pos < 0 || pos >= size)
43        throw range_error("");
44
45    if (pos == 0)
46    {
47        head = head->Next();
48        size--;
49        return;
50    }
51    node_T *prev = element_at(pos - 1);
52    if (pos == size - 1)
53    {
54        prev->DeleteNext();
55        tail = prev;
56    }
57    else
58    {
59        node_T *nxt = element_at(pos + 1);
60        prev->SetNext(*nxt);
61    }
62
63    size--;
64 }
65 template <typename T>
66 node<T> *linked_list<T>::element_at(int i)
```

```
67 {
68     node_T *nod;
69     nod = head;
70     while (i--)
71     {
72         if (!(nod->HasNext()))
73         {
74             throw range_error("");
75         }
76         nod = nod->Next();
77     }
78     return nod;
79 }
80 template <typename T>
81 T linked_list<T>::operator[](int i)
82 {
83     return element_at(i)->value;
84 }
85
86 #pragma end
```

7.1. Noexcept

El operador noexcept especifica si una función podría generar excepciones. El resultado es verdadero si el conjunto de posibles excepciones de la expresión está vacío y falso en caso contrario. El operador noexcept no evalúa la expresión. Se puede usar dentro del especificador noexcept de una plantilla de función para declarar que la función lanzará excepciones para algunos tipos pero no para otros.

Un incentivo para aplicar noexcept a funciones que no producirán excepciones es que permite a los compiladores generar un mejor código objeto y son más optimizables que las funciones que no son noexcept.

Es particularmente valioso para las operaciones de movimiento, intercambio, memoria funciones de desasignación y destructores.

7.2. Inferencia de tipo en C++ (auto, decltype decltype(auto))

- auto: deduce el tipo de una variable declarada a partir de su expresión de inicialización.
- decltype: produce el tipo de una expresión especificada.
- decltype(auto): se utiliza para declarar una función de plantilla cuyo tipo de valor devuelto dependa de los tipos de sus argumentos de plantilla, o bien para declarar una función de plantilla que contenga una llamada a otra función y devuelva el tipo de valor devuelto de la función contenida.

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.

```
1
2  template<typename R,typename... T>
3  using Function = R(*)(T...args);
4
5  template<typename R>
6      static linked_list<R> Map(linked_list<T> lst, Function<R,T> f)
7      {
8          linked_list<R> out = linked_list<R>();
9          for (int i = 0; i < lst.size; i++)
10         {
11             out.Add_Last(f(lst[i]));
12         }
13         return out;
14
15     }
```