Lenguajes de Programación
Seminario 1: C++11, C++14

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

Ciencias de la computación

Octubre, 2022

Clase genérica node

```
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);

};
```

Clase genérica linked_list

```
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;
};
```

Manejo más "inteligente" de la memoria a partir de C++11

- Introducción del principio "la adquisición de recursos es inicialización" o por sus siglas en inglés *RAII*, con lo cual se reserva o libera espacio en memoria.
- Se evita usar la memoria del heap tanto como sea posible declarando objetos en la pila.
- Si un recurso es demasiado grande para la pila, debe ser propiedad de un objeto, que sí se declara en ella y es el responsable de liberar el recurso en su destructor.
- Recolección de basura determinista. Un recurso siempre se libera en un punto conocido del programa, que puede controlar.

Elementos concretos introducidos para el manejo inteligente de memoria

Punteros Inteligentes (Smart Pointers)

Los punteros inteligentes 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.

Contenedores

Los contenedores son estructuras de datos quienes se ocupan de la gestión del espacio de almacenamiento necesario.

Inicialización

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

- **std::auto_ptr**
- **std::unique_ptr**
- **std::shared_ptr**
- **std::weak_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).

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).

Solo movimiento!!

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

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.

std::shared_ptr

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.

std::shared_ptr

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.

std::weak_ptr

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**

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.
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.

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

Para ganar en espacio y eficiencia:

- Solo se reserva memoria en el heap cuando es absolutamente necesario y se necesita asignar un gran bloque de memoria.
- Se pasa por referencia en lugar de por valor (siempre que sea posible).
- Se utilizan semánticas de movimiento en lugar de copia (siempre que sea posible).

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.

Categoría de valor

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.

Categoría de valor

lvalue

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.

Categoría de valor

rvalue

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.

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

- 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).

Constructores

- 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.

Constructores

- 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.

Sobrecargas

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

Constructores

```
#pragma constructors

linked_list(const linked_list &a) // copy ctor
{
    this->operator=(a);
}
linked_list &operator=(const linked_list &a) // copy assignment ctor
{
    size = 0;
    head = tail = nullptr;
    if (a.size == 0)
    {
        return *this;
    }
    node_T *nod = a.head;
    this->Add_Last(nod->value);
    while (nod->HasNext())
    {
        nod = nod->Next();
        this->Add_Last(nod->value);
    }
    return *this;
}
```

Constructores

```
linked_list(linked_list &&a) // move ctor
{
    head = a.head;
    tail = a.tail;
    size = a.size;
    a.head = a.tail = nullptr;
    a.size = 0;
}
linked_list &operator=(linked_list &&a) // move assignment ctor
{
    head = a.head;
    tail = a.tail;
    size = a.size;
    a.head = a.tail = nullptr;
    a.size = 0;

    return *this;
}

#pragma endregion
```

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.
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.

Constructor que permite hacer list-initialization similar a C#. for_each con expresiones lambda

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

{} v.s ()

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

Constructor que reciba un vector $< T >$

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

Destructor

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.
$\rightarrow$ 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.

Destructor

```
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;
        }
```

Raw pointer

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.

Raw pointers, ¿cuándo utilizarlos?

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.

Definición de funciones: *Add_Last*

```
#pragma implementation_of_list_members

template <typename T>
void linked_list<T>::add_element(T val)
{
    if (!size)
    {
        head = tail = new node<T>(val);
        size = 1;
        return;
    }
    tail->CreateNext(val);
    // *(tail -> next) = nod;
    tail = tail->Next();
    size++;
    if (size == 2)
    {
        head->SetNext(*tail);
    }
}
```

Definición de funciones: *Remove_Last*

```
template <typename T>
void linked_list<T>::pop_front()
{
    if (!size)
    {
        throw range_error("");
    }
    delete_at(0);
}

template <typename T>
void linked_list<T>::pop_element()
{
    if (!size)
    {
        throw range_error("");
    }
    delete_at(size - 1);
}
```

Definición de funciones: *Remove_At*

```
template <typename T>
void linked_list<T>::delete_at(int pos)
{
    if (pos < 0 || pos >= size)
        throw range_error("");

    if (pos == 0)
    {
        head = head->Next();
        size--;
        return;
    }
    node_T *prev = element_at(pos - 1);
    if (pos == size - 1)
    {
        prev->DeleteNext();
        tail = prev;
    }
    else
    {
        node_T *nxt = element_at(pos + 1);
        prev->SetNext(*nxt);
    }

    size--;
}
```

Definición de funciones: *At*

```
template <typename T>
node<T> *linked_list<T>::element_at(int i)
{
    node_T *nod;
    nod = head;
    while (i--)
    {
        if (!(nod->HasNext()))
        {
            throw range_error("");
        }
        nod = nod->Next();
    }
    return nod;
}
template <typename T>
T linked_list<T>::operator[](int i)
{
    return element_at(i)->value;
}

#pragma end
```

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.
- 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.
- Permite a los compiladores generar un mejor código objeto.
- Las funciones que son noexcept son más optimizables que las que no lo son.

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

Inferencia de tipo en C++

- 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.

Puntero a función *Function* $< R, T... >$ que devuelve un valor de tipo R y recibe un número variable de parámetros de tipo T

```
template<typename R,typename... T>
using Function = R(*)(T...args);

template<typename R>
    static linked_list<R> Map(linked_list<T> lst, Function<R,T> f)
    {
        linked_list<R> out = linked_list<R>();
        for (int i = 0; i < lst.size; i++)
        {
            out.Add_Last(f(lst[i]));
        }
        return out;

    }
```