Universidad Nacional de San Agustín, Escuela Profesional de Ingeneria de Sistemas

OOP Listas en C++

Richard Willy Alvarez Mamani

4 de septiembre de 2018

1. Introducción

El paradigma orientación objetos esta basado en la abstracción clases, y las instación de ellas objeto. Es muy fuerte la tendencia, debido a su semejanza con la realidad, y permite un agrupamiento un orden para programar y retulización y encapsulación. En este articulo, se menciona la implementación usando conceptos de OOP en c++ en una lista.

1.1. Listas

Listas, estructura de dato más simple, que permite aplicar los conceptos de OOP, en su construcción, através de clases y la modulación a nivel de clases, permite un codigo claro.

1.2. Listas Templates

Las listas template es la manera para trabajar con datos genericos, permite ingreso con diferentes tipos de datos, en esta lista se implementara con templates.

2. Plantilla

Para la implementación tendremos disponible a manera de plantilla, o interface los métodos para implementar y las clases correspondientes:
Clase MAIN aplicativo:

```
#include "node.h"
#include "list.h"

int main(){
    List<int> myfirstList;
    //your operations here

    return 0;
}
```

Clase Node ya implementadada.

```
/*
 * Node.h
 * Created on:  13/10/2014
 *  Modified on: 28/08/2018
 *  Author: Christian E. Portugal-Zambrano
 */

#ifndef NODE_H_
#define NODE_H_
#include <iostream>

template <class T>
class Node{
private:
    T data;
    Node<T>* next;

public:
    Node(T _data):data(_data),next(NULL){}
    ~Node(){}

    T getData(){
        return data;
    }

    Node<T>* getNext(){
        return next;
    }

    void setNext(Node<T>* _next){
        next = _next;
    }
};

#endif /* NODE_H_ */
```

Clase List a implementar:

```
/*
 * List.h
 *
 *  Created on: oct. 11, 2014
 *      Author: christian
 */

#ifndef List_H_
#define List_H_

#include "node.h"
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>

template <class T>
class List{
public:
   List(); // constructor
   ~List(); // destructor
   void insert(T _data); //insert an element
   void erase(T _data);    //erase and element if exist
   void clear();  //clear all the list
   void remove(int _pos); //remove element at _pos
   void reverse(); //reverse the entire list
   T at(int _pos); //get element at _pos
   bool isEmpty(); //true if is empty
   bool save(std::string); // save to a file
   bool load(std::string); // load from file
   unsigned int size(); //size of the list
   void show(); //show the list to console
   T operator[](int); //overload []
   void operator<<(T); // overload <<
private:
   Node<T>* proot;
   int Size;
};
```

3. Implementación

3.1. Clase main implementación

```
#include "node.h"
#include "list.h"
#include <iostream>
using namespace std;
int main(){
    List<int> myfirstList;
    //List<int>* myfirstList = new List<int>();
    myfirstList.insert(1);
    myfirstList.insert(2);
    myfirstList.insert(3);
    myfirstList.insert(4);
    //printf("%i\n", myfirstList.at(2));
    //myfirstList.erase(2);
    //myfirstList.remove(4);
    myfirstList.reverse();
    myfirstList.load("prueba.txt");
    //myfirstList.save("prueba.txt");
    //cout<< "hola"<< endl;
    myfirstList.show();

    return 0;
}
```

3.2. Clase Lista implementada

```
#ifndef List_H_
#define List_H_

#include "node.h"
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;
template<class T>
class List {

public:
    List(); // constructor
    ~List(); // destructor
    void insert(T _data); //insert an element
    void erase(T _data);    //erase and element if exist
    void clear(); //clear all the list
    void remove(int _pos); //remove element at _pos
    void reverse(); //reverse the entire list
    T at(int _pos); //get element at _pos
    bool isEmpty(); //true if is empty
    bool save(std::string); // save to a file
```

```
    bool load(std::string); // load from file
    unsigned int size(); //size of the list
    void show(); //show the list to console
    T operator[](int); //overload []
    void operator<<(T); // overload <<
private:
    Node<T>* proot;
    int Size;

};

template<class T>
List<T>::List() {
    proot = NULL;
    Size = 0;

}

template<class T>
List<T>::~List() {
}

template<class T>
void List<T>::insert(T _data) {
    try {

        if (Size == 0) {
            proot = new Node<T>(_data);
            Size++;
            return;

        }
        if (proot == NULL)
            return;

        //caso general
        Node<T>* aux = proot;
        int i = 0;
        while (i < Size - 1 && aux->getNext() != NULL) {
            i++;
            aux = aux->getNext();

        }
        if (i == Size - 1) {
            aux->setNext(new Node<T>(_data));
            //cout<< "inserte 2"<< endl;
            Size++;

        }
    } catch (std::bad_alloc&) {
        cout << "no inserte" << endl;
    }

}
```

```
template<class T>
void List<T>::erase(T _data) {
    if (proot == NULL)
        return;
    if (proot->getData() == _data) {
        proot = proot->getNext();
        Size--;
        return;
    }
    //caso general
    Node<T>* aux;
    for (aux = proot; aux->getNext() != NULL; aux = aux->getNext()) {
        if (aux->getNext()->getData() == _data)
            break;
    }
    if (aux->getNext() != NULL) {
        aux->setNext(aux->getNext()->getNext());
        Size--;
    }
}

template<class T>
T List<T>::at(int _pos) {
    T dato = -1;
    int i = 1;
    for (Node<T>* aux = proot; aux != NULL; aux = aux->getNext()) {
        if (i == _pos) {
            return aux->getData();
        }
        i++;
    }
    return dato;
}

template<class T>
bool List<T>::isEmpty() {
    return 0 == Size;
}

template<class T>
unsigned int List<T>::size() {
    return Size;
}

template<class T>
void List<T>::show() {

    Node<T>* aux = proot;
    while (aux != NULL) {
        printf("%i\n", aux->getData());
        aux = aux->getNext();

        //cout<< "dentro"<< endl;
```

```
    }

}

template<class T>
void List<T>::clear() {
    proot = NULL;
    Size = 0;
}

template<class T>
void List<T>::remove(int _pos) {
    if (proot == NULL)
        return;
    if (_pos == 1) {
        proot = proot->getNext();
        Size--;
        return;
    }
    //caso general
    int i = 1;
    Node<T>* aux;
    for (aux = proot; aux->getNext() != NULL; aux = aux->getNext()) {
        if (++i == _pos)
            break;
        i++;
    }
    if (i == _pos) {
        aux->setNext(aux->getNext()->getNext());
        Size--;
    }
}

template<class T>
T List<T>::operator [](int _pos) {
    T dato = 0;
    return dato;
}

template<class T>
void List<T>::operator <<(T _dato) {

}

template<class T>
bool List<T>::save(std::string filepath) {
    ofstream archivo;
    //archivo.open("prueba.txt",ios::out); //abriendo
    archivo.open(filepath, ios::out);
    if (archivo.fail()) {
        cout << "No se pudo abrir el archivo";
        exit(1);
    }
    Node<T>* aux = proot;
```

```
    while (aux != NULL) {
        archivo << aux->getData()<<endl;
        aux = aux->getNext();
    }

    archivo.close();
    return true;
}

template<class T>
bool List<T>::load(std::string filepath) {
    ifstream archivo;
    string texto;
    archivo.open(filepath,ios::in);
    if(archivo.fail()){
        cout<<"No se pudo abrir el archivo";
        exit(1);
    }
    //List<string> listAux;
    while(!archivo.eof()){//mientras no sea el final del archivo
        getline(archivo,texto);
        cout<<texto<<endl;
        //listAux.insert(texto);
    }
    archivo.close();
    //listAux.show();
    return true;
}

template<class T>
void List<T>::reverse() {
    Node<T>* sig = proot;
    Node<T>* ant = NULL;
    while (proot != NULL) {
        sig = sig->getNext();
        proot->setNext(ant);
        ant = proot;
        proot = sig;
    }
    proot = ant;
}

#endif /* List_H_ */
```

La clase node no es necesario implementarla.

4. To DO

• ¿How many constructors should be there?
tres recomendable para un mejor manejo de los objetos.
• ¿Is necessary to implement all the constructors?
Si es necesario, si queremos un mejor rendimiento y uso de los recursos.
• ¿How much memory you are using?
20bytes * elementos de la lista.
• ¿After the program nished, the memory was released?
Gracias al destructor
• ¿A constructor can be private?
Si, para el patron singleton y para una clase de constantes estaticas.
• ¿If I make a change into List, would it aect node implementation?
No, por ser moduladbles, La lista no altera a la clase Node, tienen un cierto grado de independencia.
• ¿When you implemented operator overload, have you noticed if it is really neces- sary?
No entiendo del todo sobre overload.

5. Resultados y conclusiones

Los resultados son correctas con la aplicación y evaluación de los métodos en el main.cpp implementado, se tiene 3 dudas sobre load y los dos sobre overload. Para el caso de load, conseguir que leer de una arhcivo y grabarlo a una lista. En el caso de overload no entiendo su funcionalidad.
Personalmente tengo mas dominio de java, pero para empezar en OOP necesitamos un IDE para corregir los errores o tener mayor guia para la correción. La instalación es tediosa debido a la configuración en la creación de sus proyectos.

5.1. Conclusiones

Las listas en el lenguaje de C++ resulta mas fáciles su implementación si se tiene un IDE configurado.
Las listas es un buen comienzo para apliar los conceptos de OOP en un lenguaje nuevo.
La implememtanción fue a base del lenguaje java, para comparar su sintaxis.

Referencias

[1] *Constructor privado*, https://ismaeltesisteco.wordpress.com/2013/04/21/constructor-privado/

[2] *Aprender C++*, https://www.youtube.com/watch?v=GaqgqQL3wnQ