Polimorfismo
José Miguel Huamán Cruz - CUI: 20103389 - josemiki24@gmail.com
Profesor: Cristian Lopez del Alamo
Escuela Profesional de Ciencia de la Computación
Facultad de Producción y Servicios
Universidad Nacional de San Agustín

1. Explicacion del Codigo

```
#include <iostream>

using namespace std;


class objeto //Clase Abstarcta
{
     public:
       virtual objeto* clone()=0; //Constructor
       virtual void read()=0; //Funcion Virtual Pura pero modificable en
                              //los hijos gracias a la palabra virtual
       virtual void print()   //Funcion modificable en los hijos
                              //gracias a la palabra virtual
               {
                   cout<<"soy abstracto sonso "<<endl;
               };
       virtual ~objeto() //Destructor : la palabra virtual le indica que al
                         //destruirse el objeto llame al destructor de la
                         //clase que este dentro del objeto
               {
                  cout<<"objeto muerto "<<endl;
               }
};


class Entero : public objeto //Hereda todo el contenido public de la clase objeto
{
     private:
        int m_dato; // miembro dato privado de la clase
     public:
       Entero(Entero & ent) // recibe el contenido de una clase entero
       {
          m_dato = ent.m_dato; //guarda el contenido de la clase
                               //en m_dato que guarda enteros
       }
       Entero(int a) // recibe un entero
       {
          m_dato = a; //guarda el entero en m_dato
       };

       void read()
       {
          cout <<"ingrese entero :"; // imprime en pantalla ingrese dato
          cin>>m_dato; // espera a  que ingreses por consola el dato
       }
       void print()
       {
```

```
48            cout <<"imprime entero :"<<m_dato<<endl;//imprime el contenido de m_dato
49        }
50        objeto*  clone()
51        {
52                return new Entero(*this); // saca una copia de el entero que se
53                                          //este trabajando gracias a la
54                                          //palabra this
55        }
56        ~Entero()
57        {
58            cout<<"muero el entero..."<<endl; //destruye a la clase entero
59        }
60
61 };
62
63
64
65 class Float : public objeto //clase identica a clase entero pero
66                             //solo que esta guarda floats
67 {
68      private:
69         float m_dato;
70      public:
71        Float(Float & ent)
72        {
73            m_dato = ent.m_dato;
74        }
75        Float(float a)
76        {
77            m_dato = a;
78        };
79
80        void read()
81        {
82             cout <<"ingrese float :";
83             cin>>m_dato;
84        }
85        void print()
86        {
87             cout <<"imprime float :"<<m_dato<<endl;
88        }
89        objeto*  clone()
90        {
91             return new Float(*this);
92        }
93        ~Float()
94        {
95             cout<<"muero el flotante..."<<endl;
96        }
97
98 };
99
100
101 class Lista;
102
103 class Nodo
104 {
```

```
105     friend class Lista; // la palabra friend le indica a clase nodo
106                         // que la clase lista tiene acceso a la parte
107                         // private de la clase nodo
108     private:
109       objeto * m_Dato;//guarda direccion de una clase objeto
110       Nodo   * m_pSig;//guarda la direccion de un nodo
111     public:
112       Nodo(objeto * p)//constructor que recibeun objeto por referencia
113       {
114             m_Dato = p->clone();//saca una copia y la gurada en m_Dato
115             m_pSig=0;//asigna vacio al valor de m_Psig
116       }
117 };
118
119
120 class Lista
121 {
122     typedef Nodo * pNodo; // define un alias a pNodo en vez de Nodo*
123     private:
124        pNodo m_Head;//guarda la direccion d ela cabeza de la lista
125        pNodo m_Last;//gurada la direccion del ultimo de la lista
126     public:
127       Lista()//constructor de lista
128       {
129           m_Head = m_Last =0;//asiga cero a la cabeza y cola poruqe esta vacia
130       }
131       void push_front(objeto * p)
132       {
133            pNodo nuevo = new Nodo(p); // crea un nuevo nodo
134            if(!m_Head) // si no hay nadie (si la lista esta vacia)
135            {
136               m_Head = m_Last = nuevo; // el nuevo es la cabeza y la cola ahora
137            }
138            else
139            {
140                 nuevo->m_pSig = m_Head; // el siguiente del nuevo
141                                         //tiene que ser la cabeza
142                 m_Head = nuevo;    //  y ahora la cabeza es el nuevo
143            }
144
145       }
146
147     void push_back(objeto * p)
148     {
149
150            pNodo nuevo = new Nodo(p); // crea un nuevo nodo
151            if(!m_Head) // si no hay nadie (si la lista esta vacia)
152            {
153               m_Head = m_Last = nuevo; // el nuevo es la cabeza y la cola ahora
154            }
155            else
156            {
157                 m_Last->m_pSig = nuevo; // el siguiente del ultimo es  el nuevo
158                 m_Last = nuevo;    //  el ultimo es el nuevo
159            }
160     }
161
```

```
162     void print()
163     {
164         pNodo p = m_Head;
165         while(p!=0)  // miestras p no es nulo
166         {
167             p->m_Dato->print(); // imprime el dato
168             p=p->m_pSig;  // que pase el siguiente
169         }
170 
171     }
172 
173 
174 };
175 
176 
177 
178 int main()
179 {
180       objeto *  v[ ] = {new Entero(3), new Float(5.6), new Entero(5)};
181       Lista a;
182       for(int i=0;i<3;i++)
183       {
184           a.push_back(v[i]);
185           delete v[i];
186        }
187       a.print();
188 
189       return 1;
190 }
```