# Pila Dinámica
# en lenguaje C++

por Fabián Gentile

- Clase Nodo
- Implementación clase Nodo
- Clase Pila
- Implementación clase Pila
- Métodos Clase Pila
- Uso clase Pila

## Implementación de pila y cola

A nivel genérico existen diversos modos de implementar una pila o cola.

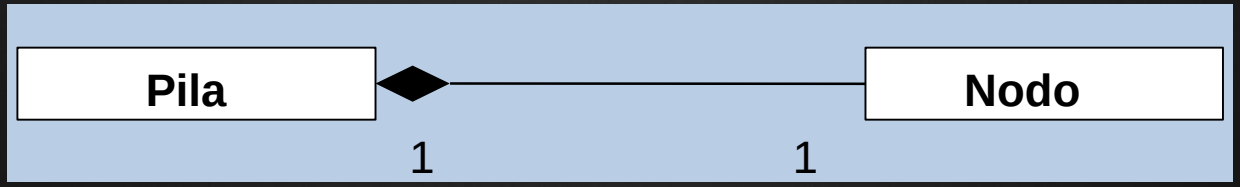
- Utilizando un array con cantidad de elementos prefijada.
- Utilizando una estructura de datos dinámica (ej.: lista enlazada).

Luego, en utilizando lenguaje C++ podemos optar por:

- Utilizar una estructura de datos determinada como elemento de la pila o cola.
- Utilizar templates (o plantillas) de modo de independizarnos del tipo de dato del elemento.(STL tema clase 3)

- Crearemos dos clases:
- una que represente la pila y otra que represente un nodo.
- A su vez la pila contendrá el puntero a un nodo: el tope de la pila

La relación entre ambas clases se puede representar como:



## Clase Nodo

Implementaremos la pila haciendo uso de una clase Nodo
y un tipo de dato predefinido

```
class Nodo{
    int   _data;
    Nodo* _next;
public:
    Nodo();
    ~Nodo();
    void setData(int  data);
    void setNext(Nodo* next) ;
    int getData();
    Nodo* getNext();
} ;
```

## Implementación clase Nodo

```
//Constructor por defecto inicializa el dato y el puntero al siguiente nodo
Nodo::Nodo(){
    _data=0;
    _next=NULL;
}
// Destructor libera la memoria borrando los nodos en cascada
Nodo::~Nodo(){
    if ( _next != NULL)
            delete this->_next;
}
```

# Implementación clase Nodo



**Setters y Getters**

```
// Seteo el dato del nodo
void Nodo::setData(int data){
    _data=data;
}
// Leo el atributo _data
int Nodo::getData(){
    return _data;
}
```

```
// Seteo el valor de _next del nodo
void Nodo::setNext(Nodo* next){
    _next=next;
}
// Leo el atributo _next
Nodo* Nodo::getNext(){
    return _next;
}
```

# Clase Pila

Implementaremos la pila cuyo atributo es un puntero de clase Nodo denominado el tope de la pila

```
#define DATAERROR -1

class PilaD{
private:
    Nodo* _tope;                // nodo tope de la pila
public:
    PilaD();                    // Constructor
    ~PilaD();                   // Destructor
    bool push(int data);        // Apilar
    bool pop(int &data);        // Desapilar
    bool empty();               // si está Vacía
    bool full();                // si está Llena
    void clear();               // elimina todos los nodos
};
```

# Método constructor y destructor

## Constructor

```
PilaD::PilaD{
        _tope=NULL;
}
```

Inicializa a _tope en NULL

## Destructor

```
PilaD::~PilaD( ){
    if(_tope!=NULL)
            delete _tope;
}
```

Si _tope no es NULL llama al destructor de Nodo y destruye en cadena todos los nodos

# Método push(apilar)

```
bool PilaD::push(int data){
Nodo* aux;                          // Nodo a crear
   if( !full() ){                   // si no está llena
            aux=new Nodo( );        // crea un nodo
            aux->setData(data);     // asigna el dato
            aux->setNext(_tope);    // engancha el nodo
            _tope=aux;              // actualiza el tope
            return true;
   }
   else{
            cout<<"pila llena"<<endl;
            return false;
   }
}
```

**Verifica si la pila no está llena y crea un nodo.**

**Le asigna su dato y el valor del puntero al siguiente nodo.**

**Actualiza la pila enganchando el nuevo nodo a la misma.**

# Método pop(desapilar)

```
bool PilaD::pop(int &data){
Nodo* aux;                          // nodo a Eliminar
   if( !empty( )){                  // si no está vacía
            aux=_tope;              // guarda el nodo tope en el auxiliar
            data=_tope->getData(); // obtiene el dato del nodo tope
            tope=_tope->getNext(); // asigna al nodo tope el sig.nodo
            aux->setNext(NULL);     // desengancha el nodo
            delete aux;             // libera la memoria del nodo
            return true;
   }
   else{
            cout<<"pila vacia"<<endl;
            data=DATAERROR;
            return false;
   }
}
```

**Verifica si la pila no está vacía.**

**Le asigna al nodo auxiliar el nodo tope.**

**Obtiene el dato.**

**Asigna al nodo tope el siguiente nodo.**

**Desengancha en nodo a eliminar.**

**Devuelve la memoria utilizada.**

# Método full y empty

## Verifica si no está llena la pila

```
bool PilaD::full(){
Nodo* aux = new Nodo();
   if (aux != NULL){
            delete aux;
            return false;
   }
   return true;
}
```

Se crea un nodo auxiliar para verificar si hay memoria disponible.

Si hay memoria se destruye el nodo y devuelve false.

En caso contrario devuelve true.

## Verifica si no está vacía la pila

```
bool PilaD::empty(){
   return (_tope==NULL);
}
```

devuelve true si está vacía

# Método clear(limpiar)

```
void Pila::clear( ){
   if (_tope) delete _tope;
   _tope=NULL;
}
```

**Elimina todos los nodos de la pila e inicializa el tope de la misma.**

**(Al eliminar un nodo se invoca al destructor de la clase Nodo que elimina en cascada todos los nodos)**

**Todos los datos se pierden en esta operación.**

# Pila Dinámica - Ejemplo Uso

```
#include “PilaD.h”
void main(){
PilaD* p=new PilaD();
int dato;
    cout<<“Apilando 5 números:”<<endl;
    for(int i=0;i<5;i++)
        p->push(i);

    cout<<“Desapilando toda la pila:”<<endl;
     while(!p->empty()){
        p->pop(dato);
        cout<<“dato:”<<dato<<endl;
    }
    delete p;
    cin.get();
}
```

crea un objeto de clase pila

apila 5 números enteros

mientras la pila no esté vacía

desapila un dato

destruye el objeto pila

La salida es:
Apilando 5 números
Desapilando toda la pila
dato: 4
dato: 3
dato: 2
dato: 1
dato: 0

# FIN