

Tipos de Datos Abstractos

Desde especificaciones a código C++

1. Introducción

En la búsqueda de una estructura modular basada sobre tipos de objetos, los tipos de datos abstractos (TDA) proveen un mecanismo de descripción de alto nivel, independiente de implementaciones particulares.

Un tipo de dato abstracto es un conjunto de valores sobre los que se aplica un conjunto de operaciones que cumplen determinadas propiedades.

El calificativo abstracto responde al hecho de que los valores de un tipo pueden ser manipulados mediante sus operaciones, si se saben las propiedades que éstas cumplen, sin que sea necesario ningún conocimiento adicional sobre el tipo.

1.1. Especificación de un TDA

Los TDA son la pieza central en las metodologías de programación actuales, por ello requieren que se especifiquen (o describan) adecuadamente.

La especificación debe dar una clara y precisa descripción del problema, de **lo que el sistema debe hacer** y no de cómo lo debe hacer, es decir, se abstrae de los detalles de la implementación.

Existen ciertas características relevantes que toda buena especificación debe satisfacer, y es que estas descripciones deben ser:

- **precisas y no ambiguas:** las especificaciones informales, escritas en lenguaje natural, pueden conducir a interpretaciones erróneas.
- **completas:** debe definirse todo concepto que se use, y todas las suposiciones deben ser explícitamente establecidas.
- **no sobre-especificadas:** se debe especificar sin dar detalles de la representación del tipo.

Para describir el comportamiento de un TDA, es obvio que el lenguaje natural no es una buena opción dada su falta de precisión. Por lo tanto, es conveniente una notación formal. Una especificación es formal si está formulada en un lenguaje formal, un lenguaje cuya sintaxis y semántica son establecidas explícitamente. La especificación algebraica es un poderoso formalismo para definir objetos, clases de objetos y sus operaciones en forma abstracta e independiente de la implementación. Un álgebra es un conjunto de valores sobre los que se aplican operaciones y esta definición es similar a la noción de tipos de datos abstractos y por este motivo normalmente se estudian los TDAs dentro del marco de las álgebras heterogéneas.

Una especificación algebraica se compone de la signatura del tipo y un conjunto de axiomas.

La **signatura de un tipo** es un par formado por el conjunto de sorts (S) y los símbolos de operaciones (F):

$$\Sigma = \langle S, F \rangle$$

Luego, la **especificación básica** de un tipo consta de la signatura del tipo y el conjunto de axiomas

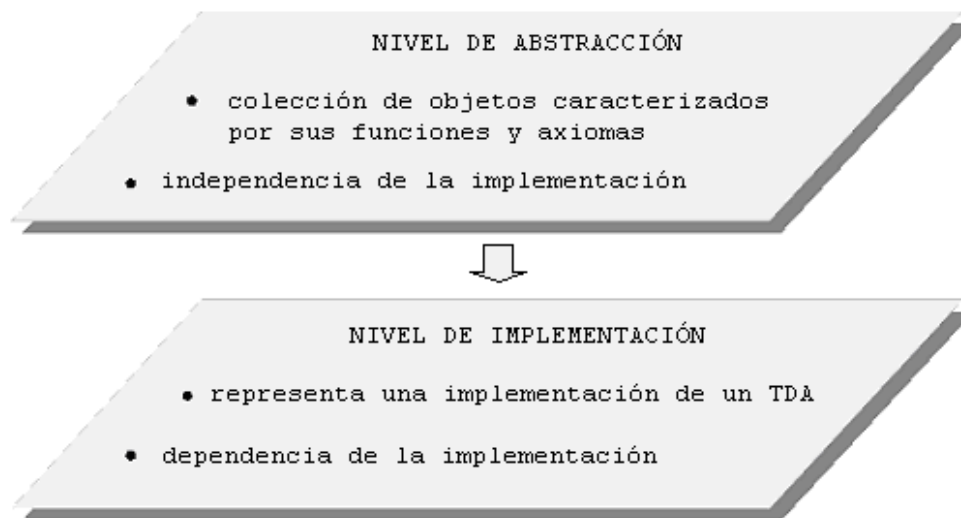
$$\Sigma + \text{Axiomas}$$

Para especificar TDAs, utilizaremos el lenguaje de especificación algebraica NEREUS. Una descripción detallada puede encontrarse en: <http://aydalgor.alumnos.exa.unicen.edu.ar/apuntes-de-teoria>

Una introducción a las especificaciones algebraicas y su relación con el paradigma de orientación a objetos puede consultarse en Meyer (1997) (capítulo 6).

1.2. Implementación de un TDA

La implementación de un TDA es la traducción de su especificación a un lenguaje de programación.



Un TDA es un concepto matemático, su especificación es libre de cambios, usando un término de la ciencia de la computación teórica, decimos que es *aplicativa*. Las operaciones de un TDA son modeladas como funciones matemáticas. Por ejemplo, la operación *poner* de las pilas es modelada por la función:

`poner: Pila * Natural --> Pila`

especificando una operación que retorna una nueva pila, en vez de cambiar la pila existente.

La *implementación de un TDA* es su versión orientada al computador. La implementación introduce una visión imperativa, las operaciones pueden cambiar los objetos. Por ejemplo, la operación *poner* aparecería como una función que toma un argumento de tipo *Natural* y modifica la pila poniendo el nuevo elemento en el tope en vez de producir una nueva pila.

¿Cómo implementamos un TDA?

La implementación de un TDA requiere:

1. La especificación de un TDA
2. La elección de una representación: para un tipo de dato pueden existir varias alternativas de implementación.
3. La traducción de las funciones a la representación elegida en la forma de un conjunto de métodos, cada uno implementando una de las funciones en términos de la representación, satisfaciendo la especificación.

Por lo tanto, para una única especificación de un TDA se pueden construir diversas implementaciones y así seleccionar la más adecuada en cada contexto de uso.

2.1.1. Tipos

Un tipo es una colección de objetos caracterizados por sus funciones, axiomas y precondiciones. Por ejemplo, el tipo Pila es el conjunto de todas las posibles pilas. Un tipo de dato abstracto tal como la pila no es un objeto (una pila en particular) sino una colección de objetos (la colección de todas las pilas).

Un objeto perteneciente al conjunto de objetos descriptos por una especificación de un TDA se llama una instancia de un TDA. Por ejemplo, una pila específica que satisface las propiedades del TDA Pila será una instancia de Pila.

La definición de un TDA comienza por establecer qué objetos intervienen en su definición, es decir, el conjunto de tipos. Cada tipo representa el conjunto de valores que caracteriza al tipo en definición.

En la especificación del TDA Pila el conjunto de valores que caracteriza al tipo es {pila, boolean, elemento}, lo cual forma parte de la signature del tipo.

2.1.2. Funciones

En la signature de una especificación las funciones sólo listan sus argumentos y el tipo de resultado:

| | | |
|------------------|--|--|
| crear: | $--> \text{Pila}$ | $\text{crear} () = \boxed{}$ |
| poner: | $\text{Pila} * \text{elem} --> \text{Pila}$ | $\text{poner} (\boxed{}, \text{—}) = \boxed{}$ |
| elemento: | $\text{Pila} (p) --> \text{elem}$ $\text{pre: not esvacía} (p)$ | $\text{elemento} (\boxed{}) = \text{?}$ |
| sacar: | $\text{Pila} (p) --> \text{Pila}$ $\text{pre: not esvacía} (p)$ | $\text{sacar} (\boxed{}) = \boxed{}$ |
| esvacía: | $\text{Pila} --> \text{Boolean}$ | $\text{esvacía} (\boxed{?}) = \text{boolean}$ |

- **crear:** devuelve una pila vacía.
- **poner:** devuelve una nueva pila con un elemento extra en el tope.
- **sacar:** devuelve una nueva pila sin el elemento tope de la pila original, si la pila no está vacía.
- **elemento:** consulta sobre el elemento tope, si la pila no está vacía.
- **esvacía:** indica si la pila está vacía.

Precondiciones

Algunas funciones no son aplicables a todos los posibles elementos del dominio. Éste es el caso de las operaciones *sacar* y *elemento*: no se puede sacar un elemento de una pila vacía y una pila vacía

no tiene elemento tope. Debemos restringir el dominio de tales operaciones siendo éste el rol de las precondiciones:

sacar: Pila (p) --> Pila

pre: not esvacía (p);

elemento: Pila (p) --> elem

pre: not esvacía (p);

La **precondición** es una expresión booleana que restringe el dominio de aplicación de las funciones. En este ejemplo, la precondición para las operaciones *sacar* y *elemento* expresa que la pila (argumento p) no debe ser vacía.

Clasificación de funciones

Las funciones de un tipo se clasifican de la siguiente manera:

- **constructoras:** devuelven un valor cuyo tipo es igual al tipo que se está especificando:
 - **constructoras básicas o constructoras generadoras:** subconjunto mínimo de las operaciones constructoras que permiten generar, por aplicaciones sucesivas, todos los valores del tipo que queremos especificar. En la pila: *crear* y *agregar*.
 - **constructoras modificadoras:** constructoras que no forman parte del conjunto de constructoras básicas. En la pila: *sacar*.
- **observadoras:** devuelven un valor cuyo tipo es distinto al tipo que se está especificando. En la pila: *esvacía* y *elemento*.

2.1.3. Axiomas

Los axiomas declaran las propiedades del tipo especificado. Los axiomas se construyen de la siguiente manera:

- Identificar el conjunto de operaciones constructoras básicas
- Especificar el resto de las operaciones en función de cada una de las constructoras básicas.

En el caso de la pila, las operaciones constructoras son *crear* y *poner*. Luego para cualquier *e* de tipo *elem*, y cualquier *p* de tipo *Pila*, los axiomas que describen la semántica (el comportamiento) de las restantes operaciones son:

- | | | |
|---|---|--|
| <p>(1) elemento (poner (p, e)) = e</p> <p>(2) sacar (poner (p, e)) = p</p> | } | <p>Proveen una descripción concisa de la propiedad fundamental de las pilas (LIFO) en términos matemáticos, sin recurrir a razonamientos imperativos o a una representación particular. El primer axioma establece que el tope de una pila es el último elemento ingresado. El segundo axioma establece que si sacamos el tope de una pila obtenemos la pila anterior al ingreso del último elemento. En este caso particular, ambas operaciones se especifican sólo sobre la operación constructora <i>poner</i>, ya que el dominio de aplicación de esta función está restringido a pilas no vacías, especificado por la precondición.</p> |
| <p>(3) esvacía (crear ()) = true</p> <p>(4) esvacía (poner (p, e)) = false</p> | } | <p>Especifican cuando una pila es vacía y cuando no: la pila resultante de la función <i>crear</i> es vacía y la pila resultante de poner un elemento a una pila existente (vacía o no) es no vacía.</p> |

Los axiomas son predicados (en el sentido de la lógica) que expresan que cierta propiedad es siempre verdadera para cualquier posible valor de p y de e .

2.1.4. Términos del álgebra

Mediante la aplicación sucesiva y correcta de las funciones de una signatura se pueden construir términos del álgebra. Los siguientes son algunos ejemplos de términos sobre la signatura del tipo de dato abstracto *Pila[Entero]* :

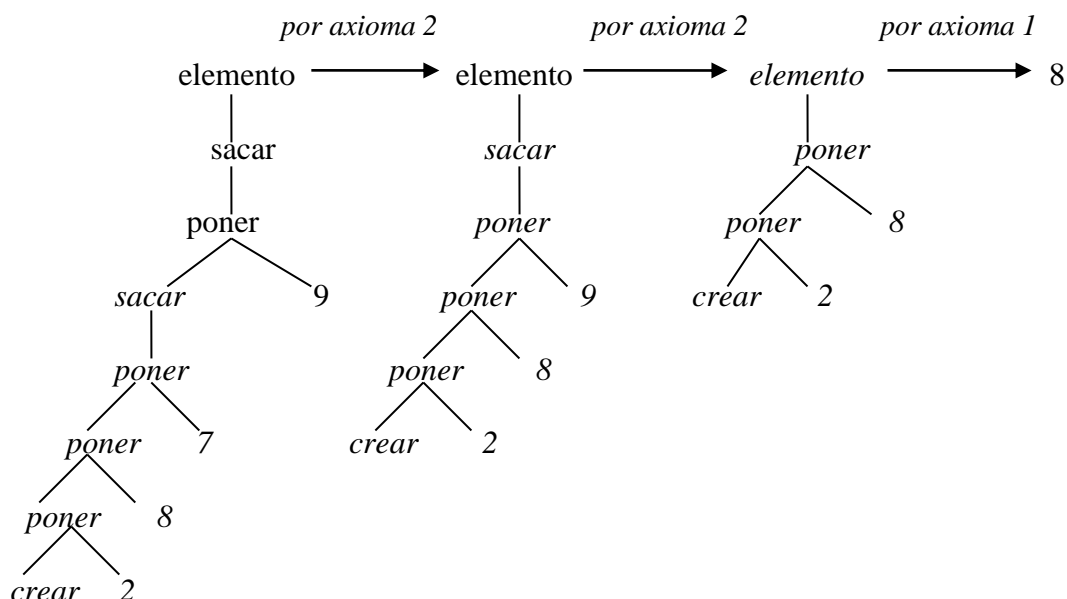
1. `crear()` -- *pila vacía*
2. `poner(crear(), 2)` -- *pila con un solo elemento*
3. `poner(poner(crear(), 2), 6)` -- *pila con dos elementos, 2 y 6, siendo 6 el tope de la pila*
4. `elemento(poner(poner(crear(), 2), 6))` -- *entero 6 (tope de la pila)*
5. `poner(poner(poner(crear(), 2), 6), 1)`
6. `sacar(poner(poner(poner(crear(), 2), 6), 1))`
7. `poner(sacar(poner(poner(poner(crear(), 2), 6), 1)), 8)`
8. `elemento(poner(sacar(poner(poner(poner(crear(), 2), 6), 1)), 8))`
9. `esvacía(sacar(poner(poner(poner(crear(), 2), 6), 1)))` -- *falso*

Como se observa en los ejemplos, las operaciones definidas en la especificación de un TDA permiten construir términos complejos. Estos términos pueden ser simplificados rescribiéndolos por medio de la aplicación sucesiva de los axiomas de la especificación.

Por ejemplo, el término

`elemento (sacar (poner (sacar (poner (poner (poner (crear() , 2) , 8) , 7)) , 9)))`

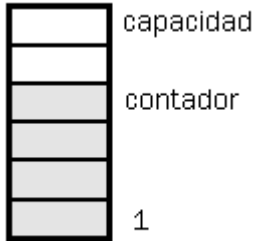
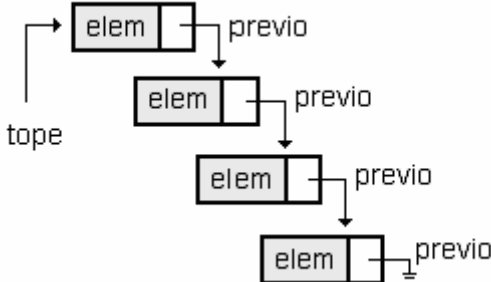
es simplificado mediante la aplicación sucesiva de los axiomas 2, 2 y 1, obteniendo el número entero 8.



2.2. Implementación del TDA Pila

Para obtener la clase Pila debemos:

- proveer la especificación del TDA Pila (ver sección 2.1),
- decidir sobre su representación física,

| Representaciones posibles | |
|--|---|
| <p>Secuencial: a través de una representación de arreglo y un contador cuyo valor varía entre 0 (pila vacía) y capacidad (tamaño del arreglo).</p> |  |
| <p>Vinculada: almacena cada elemento de la pila en una estructura con dos campos: elem representando el elemento, y previo que contiene un puntero al elemento colocado previamente en la pila.</p> |  |

- implementar las funciones del tipo Pila en base a la representación física elegida: es decir, traducir las funciones a un conjunto de métodos que satisfacen los axiomas y las precondiciones.

2.2.1. Implementación en C++

En C++ los tipos son definidos en términos de clases. Una vez que se ha definido una clase, su nombre pasa a ser un especificador de un nuevo tipo de dato creado por el usuario permitiendo crear instancias de dicho tipo usando el nombre de la clase.

Una definición de una clase describe las características de todos los objetos declarados a ser de la clase dada. Las clases pueden tener como miembros tanto datos como funciones, pueden incluir partes públicas, privadas y protegidas, lo que permite a determinadas partes, generalmente las variables de la clase, estar ocultas a todas las funciones, excepto a las funciones miembro de la clase. Todas las variables y funciones declaradas después de la palabra clave *public*, son accesibles en todas las otras funciones del programa. Por defecto, los datos y las funciones son privados a la clase.

La sintaxis de una clase es la siguiente:

```

class nombre_clase
{
    // datos y funciones privados

public:
    // datos y funciones públicos
};

```

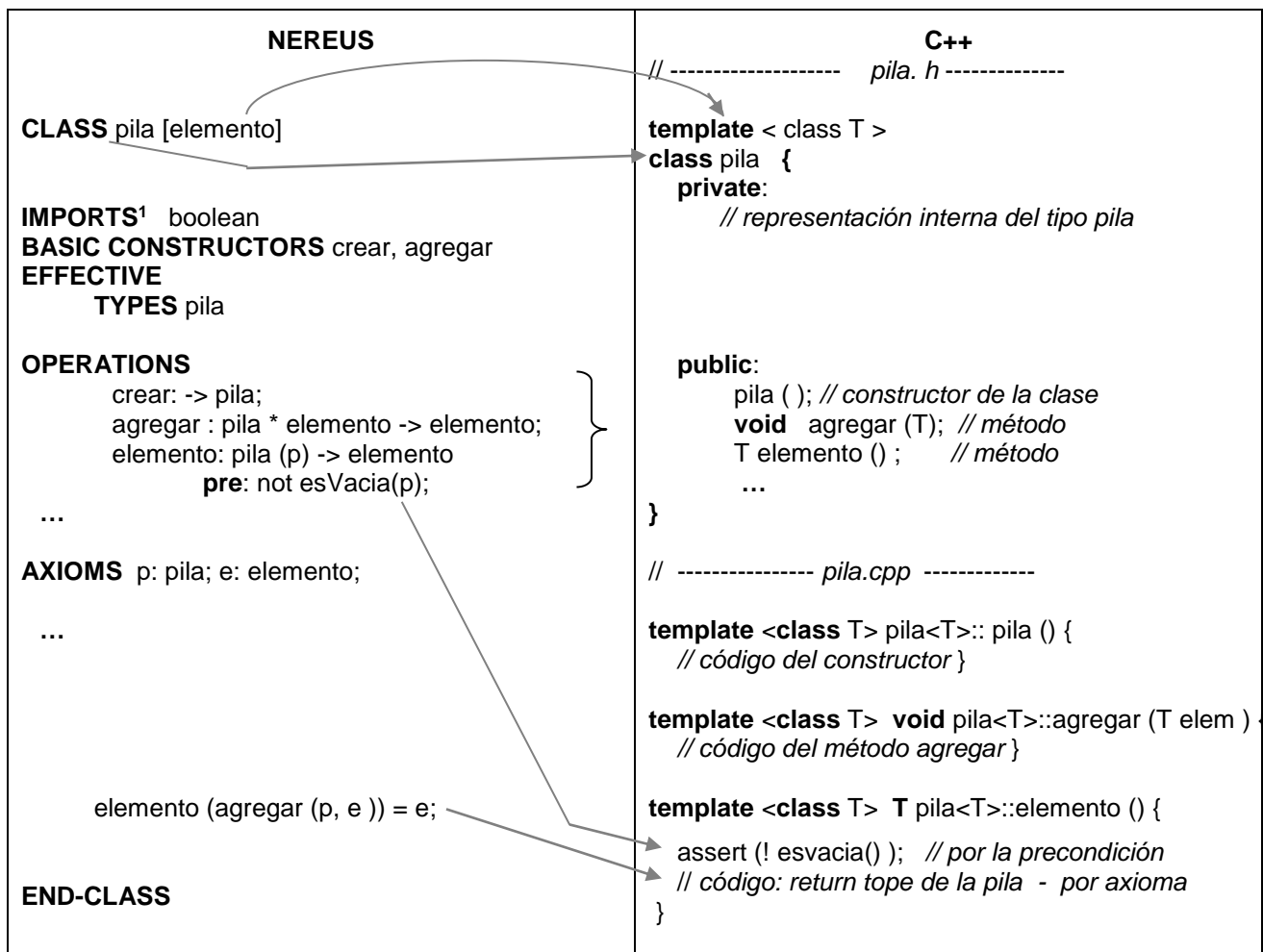
Una clase genérica o clase template, permite definir un patrón para una definición de clases. Permite la creación de objetos por medio de la parametrización. La sintaxis de una clase template es la siguiente:

```

template <lista-argumentos-template>
class nombre-clase
{ ... };

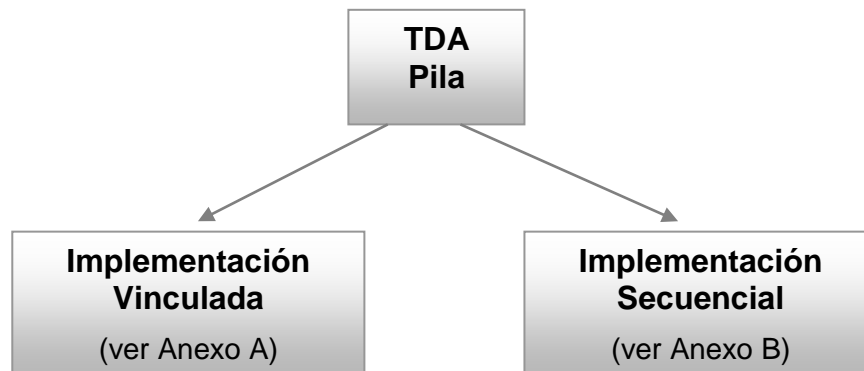
```

Correspondencia entre especificación e implementación:



¹ Por cada tipo no básico listado en la cláusula **IMPORTS** de la especificación, en la implementación se debe incluir el archivo que contiene la declaración del tipo mediante la directiva al preprocesador **#include**. En el ejemplo no es necesario por ser **bool** un tipo básico del lenguaje C++.

Como se mencionó en la sección anterior, el TDA pila puede ser implementado de maneras diferentes. En los anexos A y B se muestran dos posibles implementaciones de la pila:



Bibliografía

Bertrand Meyer (1997). *Object-oriented Software Construction*. Segunda Edición. Prentice Hall PTR.

Anexo A – Implementación del TDA Pila en C++

Implementación parametrizada utilizando una lista vinculada.

```
template <class T>
class Pila
{
    struct Nodo {
        T elem;
        Nodo * previo;
    }
    Nodo * tope;

public:
    Pila();
    ~Pila();
    void poner(const T & elemento);
    bool sacar();
    const T& elemento() const;
    bool esVacia() const;
};

template <class T>
Pila<T>::Pila() {
    tope = NULL;
}

template <class T>
void Pila<T>:: poner(const T & elemento) {
    Nodo *aux = new Nodo;
    aux->elem = elem;
    aux->previo = tope;
    tope = aux;
}

template <class T>
const T & Pila<T>:: elemento() const {
    assert (!esvacia());
    return tope->elem;
}

template <class T>
bool Pila<T>:: sacar() {
    if (ultimo != NULL) {
        Nodo* aux = tope;
        tope = tope->previo;
        delete aux;
        return true;
    }
    else
        return false;
}
```

```
template <class T>
bool Pila<T>:: esvacia() const{
    if (ultimo == NULL)
        return true;
    else
        return false;
}

template <class T>
Pila<T>::~~Pila(){
    Nodo *aux=tope;
    while (tope != NULL )
        {aux=tope;
        tope=tope->previo;
        if (aux != NULL )
            delete (aux);
        }
}
```

Anexo B – Implementación del TDA Pila en C++

Implementación parametrizada utilizando un arreglo.

```
template <class T>
class Pila
{
    T * p;
    unsigned int contador, capacidad;

public:
    Pila(unsigned int capacidad);
    ~Pila();
    bool poner(const T& elemento);
    const T& elemento() const;
    bool sacar();
    bool esvacia() const;
};

template <class T>
Pila<T>::Pila(unsigned int capacidad) {
    p= new T [capacidad];
    contador= 0;
    this->capacidad=capacidad;
}

template <class T>
bool Pila<T>:: poner(const T& elemento) {
    if (contador < capacidad) {
        p[contador++]= elem;
        return true; }
    else
        return false;
}

template <class T>
const T& Pila<T>:: elemento() const {
    assert(!esvacia());
    return( p[contador-1] );
}

template <class T>
bool Pila<T>:: sacar()
{ if (contador != 0) {
    contador-- ;
    return true; }
    else
        return false;
}
```

```
template <class T>
bool Pila<T>:: esvacia() const{
    if (contador == 0)
        return true;
    else
        return false;
}

template <class T>
Pila<T>::~~Pila() {
    delete [] p;
}
```