

Templates y STL

Templates: Leandro Lucarella

STL: Gonzalo Merayo

Revisión: 1. Lucarella, Merayo
2. Leandro H. Fernández

Templates y STL	1
Introducción	1
Funciones parametrizadas	1
Consejo	2
Nota	2
Instanciación	2
Organización en archivos	3
Clases parametrizadas	3
Importante	6
Especialización	6
STL	7
Namespace std	8
String	8
Contenedores	8
Se almacenan siempre copias	9
No usar polimórficamente	9
Acceso a los contenedores básicos	9
Iteradores	10
Iteradores const	10
Contenedor asociativo	10
Algoritmos	10
Input Output	11

Introducción

Los *templates* (plantillas) proveen a C++ la posibilidad de realizar [Programación Genérica](#). Lo que permite programar algoritmos paramétricos, utilizando (principalmente) tipos como parámetros, de manera tal de que un algoritmo pueda ser utilizado intacto para distintos tipos de datos, que comparten una interfaz común.

A simple vista pareciera que esto no ofrece ninguna ventaja sobre la programación orientada a objetos, con la cual se puede lograr el mismo efecto a través de una jerarquía de clases (o interfaces), razón por la cual también suele llamarse a las *templates* **polimorfismo estático**.

Además las *templates* hacen prácticamente obsoletas las funciones *macro* del precompilador, ya que al ser resueltas en tiempo de compilación son tan eficientes como estas, pero al ser parte del lenguaje C++, pueden hacer uso de su estado interno y son, entre otras cosas, *type-safe*.

STL significa *Standard Template Library* (Biblioteca estándar de plantillas, o *templates*). A la biblioteca estándar de C, C++ agrega una biblioteca estándar de tipos y algoritmos parametrizados, entre ellos búsqueda binaria (y otros tipos de búsquedas, ordenamiento, máximo, mínimo, reemplazo, contenedores, *strings*, números complejos, *streams*, etc).

Funciones parametrizadas

C++ permite parametrizar funciones de varias maneras, veamos algún ejemplo:

```
template < typename Tipo >
Tipo min(const Tipo& x, const Tipo& y)
{
    return x > y ? y : x;
}
```

Consejo

Es muy recomendable dejar espacios entre los "corchetes angulares" (< y >) porque en el caso de anidar *templates*, si no se hace esto, el compilador los va a confundir con el operador de *shift*: >>, por ejemplo en este caso `std::map<int, std::list<double>>`, al final nos queda el operador >> y el compilador a veces da un mensaje de error muy extraño.

La palabra clave `template` permite declarar (y/o definir) una función parametrizada. A continuación se indican los parámetros de la *template* (no confundir con los parámetros de la función). Para indicar que toma como parámetro un **tipo** se pueden utilizar las palabras reservadas *typename* o *class* indistintamente (tienen el mismo significado **en este contexto**). Es decir que podría haber escrito `template < class T >` y el resultado sería el mismo. Recomendamos utilizar la palabra *typename* para enfatizar que el parámetro no tiene que ser necesariamente una clase, puede ser cualquier otro tipo nativo del compilador (como `int`, `char` o incluso un `enum`). Finalmente `Tipo`, es el nombre de nuestro parámetro.

A continuación sigue la declaración y definición de una función común y corriente, con la salvedad de que utilizaremos `Tipo` como si fuera el nombre de un tipo existente (recordar que es un parámetro, podría no existir ningún tipo llamado `Tipo`).

Ahora es posible utilizar la función `min()` con cualquier tipo, por ejemplo `int` o `double` o una clase propia.

Nota

¿Realmente se podrá utilizar cualquier tipo? ¿O nuestra función impondrá alguna restricción?

Instanciación

Los templates en realidad no existen (*there is no spoon*), no al menos hasta que son **instanciados**. Esto significa que una *template*, sin tener el tipo definido, no puede generar código, porque es imposible para el compilador generar código que utiliza tipos de los cuales todavía no conoce el tamaño (en el caso de la función `min()`, cuántos bytes debería apilar en el *stack* al llamarse la función para pasar los parámetros?). Es por esto que si una *template* no se usa, **no genera código**, no existe. De esto se desprenden varios fenómenos particulares, a saber:

- Debe haber un mecanismo para instanciar *templates*.
- Las templates deben estar **definidas** (no solo declaradas) en todos los archivos que la usen (veremos esto en más detalle más adelante).

Para **instanciar** una *template* hay dos formas. La más común es simplemente utilizarla, por ejemplo:

```
int i = 5;
int j = min(i, 7);
```

Instancia la template `int min(const int&, const int&)`, es decir, instancia nuestra template con `Tipo = int`. Esto **genera código**, el código para la función `int min(const int&, const int&)`. Si luego escribimos:

```
double d = min(1.0, 4.2);
```

Nuevamente se instancia la *template*, esta vez con `Tipo = double`, generando **otra función**: `double min(const double&, const double&)` (recordar que en C++ se permite la sobrecarga de funciones, funciones con el mismo nombre pero distinta cantidad y/o tipo de parámetros, por lo que una función en realidad es identificada por su nombre más sus parámetros). Es decir, nuestro código objeto ahora tiene **dos** funciones `min()`. Y antes de empezar esta sección (antes de utilizarla), no tenía **ninguna**. Qué pasa si ahora escribimos en nuestro programa:

```
int k = min(i, j);
```

No se crea más código del que ya había, se utiliza la *template* ya instanciada con `Tipo = int`. También es posible instanciar una *template* explícitamente:

```
int l = min< float >(i, j);
```

Si bien `min()` toma como parámetros dos `int`, se instancia la *template* `float min(const float&, const float&)`, ya que se especificó explícitamente que `Tipo = float`. Al igual que si hubiera utilizado una función no parametrizada, el compilador se encargará de *castear* los `int` a `float` antes de pasarle los parámetros a la función.

⚠Atención!

Es muy importante que se comprenda este concepto antes de continuar con la lectura.

Organización en archivos

Como se comentó anteriormente, como el código de una *template* no es generado hasta que esta no se instancia, y como, generalmente, la instanciación viene asociada al uso, es necesario asegurarse que la **definición** de una *template* esté disponible para todos los archivos en donde ésta se use. Por lo tanto la práctica más usual es tanto **declarar** como **definir** las *templates* en los `.h`. En nuestro ejemplo, esto puede hacerse directamente poniendo el primer fragmento de código en un `.h`, o bien separando **declaración** y **definición**, pero siempre dejando todo en el `.h`:

```
template < typename Tipo >
Tipo min(const Tipo& x, const Tipo& y);

template < typename Tipo >
Tipo min(const Tipo& x, const Tipo& y)
{
    return x > y ? y : x;
}
```

Para una sola función (y tan simple), esto parece absurdo, pero cuando se declaran varias *templates* en un `.h`, y más aún cuando su implementación no es trivial, puede ser mucho más claro declarar todas las funciones primero, y definir las luego (incluso separando definición y declaración en archivos `.h` diferentes, asegurándose que al incluir el archivo con las declaraciones, también se incluya el de definiciones). Esto cobra aún más sentido cuando se parametrizan clases, ya que si no se separan declaración de definición, todos los métodos de la clase pasan a ser *inline* implícitamente.

Clases parametrizadas

La parametrización de clases es muy útil, en especial al desarrollar *contenedores* (clases que sirven para almacenar otros objetos, como listas, vectores, tablas de hash, etcétera).

Veamos un ejemplo clásico (completo) de cómo encapsular un array estático parametrizándolo:

```
#include <iostream>
#include <algorithm>
#include <cassert>
#include <cstdlib>
```

```
template <
    typename Tipo,          // Tipo que almacena el array
    size_t Tamanio = 5 > // Parámetro numérico y por omisión
struct Array
{
    // Tipos y constantes: para facilitar la programación genérica
    typedef Tipo value_type;
    enum { MAX_SIZE = Tamanio };
    // Constructores
    Array() {} // default, inline
    Array(const Tipo* array, size_t tamanio);
    // Acceso a un elemento a través de un índice para escritura
    Tipo& operator[] (size_t indice);
    // Acceso a un elemento a través de un índice para lectura
    const Tipo& operator[] (size_t indice) const;
    // Acceso a un elemento a través con chequeo de "bordes" (bound-check)
    Tipo& at(size_t indice);
    const Tipo& at(size_t indice) const;
    // Obtención de tamaño máximo
    size_t capacity() const { return MAX_SIZE; } // Este método es "inline"
    // Conversión a un array de otro tipo
    template < typename OtroTipo >
    Array< OtroTipo, Tamanio > as() const;
    // Array estático
    Tipo _array[Tamanio];
};

template < typename Tipo, size_t Tamanio >
Array< Tipo, Tamanio >::Array(const Tipo* array, size_t tamanio)
{
    for (size_t i = 0; i < tamanio; ++i) _array[i] = array[i];
}

template < typename Tipo, size_t Tamanio >
Tipo& Array< Tipo, Tamanio >::operator[] (size_t indice)
{
    return _array[indice];
}

template < typename Tipo, size_t Tamanio >
const Tipo& Array< Tipo, Tamanio >::operator[] (size_t indice) const
{
    return _array[indice];
}

template < typename Tipo, size_t Tamanio >
Tipo& Array< Tipo, Tamanio >::at(size_t indice)
{
    assert(indice < MAX_SIZE);
    return _array[indice];
}

template < typename Tipo, size_t Tamanio >
const Tipo& Array< Tipo, Tamanio >::at(size_t indice) const
{
    assert(indice < MAX_SIZE);
    return _array[indice];
}
```

```
// Para esto, Tipo debe ser implícitamente convertible a OtroTipo
template < typename Tipo, size_t Tamano >
template < typename OtroTipo >
Array< OtroTipo, Tamano > Array< Tipo, Tamano >::as() const
{
    Array< OtroTipo, Tamano > otro;
    std::copy(_array, _array + Tamano, otro._array); // algoritmo de STL
    return otro;
}

int main(int argc, char* argv[])
{
    typedef Array< int > ArrayInt; // Para escribirlo más fácil (5 elem.)
    typedef Array< char*, 10 > ArrayCharp; // con 10 elementos
    assert(argc > ArrayInt::MAX_SIZE);
    ArrayInt arrint;
    ArrayCharp args(argv+1, argc-1);
    for (size_t i = 0; i < argc-1; ++i)
    {
        std::cout << i << ": " << args.at(i); // con bound-check
        // arrint lo accedemos con chequeo de bordes, args no es
        // necesario porque ya chequeamos bordes en la línea anterior.
        arrint.at(i) = atoi(args[i]);
        std::cout << " -> " << arrint[i] << "\n"; // sin bound-check
    }
    // convertimos a array de double
    Array< double > arrdbl = arrint.as< double >();
    std::cout << "Con doubles:\n";
    std::cout.precision(2);
    for (size_t i = 0; i < arrdbl.capacity(); ++i)
    {
        std::cout << i << ": " << std::fixed << arrdbl[i] << "\n";
    }
    return 0;
}
```

Analicemos el ejemplo por partes. Primero observamos un tipo de parámetro para una *template* nuevo: `size_tTamano = 5`. Hay dos conceptos nuevos en este parámetro: el tipo (`size_t`) y el valor por omisión (`= 5`). Un *template* puede recibir también como parámetro cualquier tipo nativo del compilador, y su valor puede ser utilizado como una constante resuelta en tiempo de compilación dentro de la *template* (esto permite utilizarla como tamaño para un array estático, como vemos en el ejemplo).

Luego podemos ver un `typedef` y un `enum` con un comentario que indica que facilitan la programación genérica, esto excede el objetivo de este documento pero pueden ampliar buscando en Internet o en la bibliografía de la materia. Esta técnica es ampliamente utilizada por la STL.

A continuación se declaran un par de constructores, siendo el constructor *default inline* (es decir, el compilador hará lo posible por no generar una llamada a función cuando se utilice, en este caso como es vacío, seguramente podrá hacerlo). El método `capacity()` también es *inline*.

Luego se declaran dos variantes del `operator[]`, una para ser utilizado de forma constante (para lectura solamente) y otro para modificación, acompañado cada uno por una versión (`at()`) que incluye un rudimentario chequeo de bordes (*bound-checking*) implementado via la *macro* `assert()` de C, para asegurarse de no pisar memoria si nos vamos de los límites del array. Esto también se incluye a modo de ejemplo por ser una técnica utilizada por los contenedores de la STL (sólo que éstos utilizan excepciones para reportar errores).

Finalmente nos encontramos con un método parametrizado (`as()`), dentro de una clase parametrizada, que sirve para realizar una conversión genérica de un tipo de array a otro. Ambos

arrays tienen el mismo tamaño, pero contienen tipos de datos arbitrariamente diferentes (mientras la conversión implícita sea posible).

Luego nos encontramos con el único atributo de esta clase, que utiliza ambos parámetros, el Tipo y el Tamaño (no se utilizó control de acceso, `private`, por motivos didácticos).

Después de la declaración de la clase (struct en este caso, por simplicidad), están todas las definiciones de los métodos no *inline*. Se debe prestar atención a la definición del método **as()** que tiene dos particularidades interesantes:

1. La forma de definir un método parametrizado de una clase parametrizada (como se puede observar, esto requiere de dos palabras clave `template`).
2. La utilización de un algoritmo de la STL para copiar los datos de un array a otro (como adelanto de los algoritmos de la STL).

El resto del programa es una función **main()** que demuestra las capacidades de nuestra clase genérica. Primero muestra lo conveniente que es la utilización de `typedef` al utilizar *templates*, además de demostrar la instanciación de una template con un parámetro por omisión. Luego muestra la utilidad de haber definido el enum con `MAX_SIZE`, porque de otra forma, para escribir el **assert()**, hubiera sido necesario ver cual es el valor del parámetro por default de la template y ponerlo a mano. Esto además de ser engorroso, hace el código inmantenible, porque si un día el autor de la *template* cambia ese valor por default, introduce un *bug* en nuestro código (o hay que estar pendiente todo el tiempo de los cambios). Después se instancia un objeto de la clase `Array<char*, 10>`, inicializándolo con el array `argv` utilizando el constructor.

Importante

Notar que el nombre de la clase es `Array<char*, 10>`, no `Array` solo, `Array` es una *template*, no una clase.

Luego se copian los valores de un array a otro (convirtiendo los valores de texto a int), haciendo *bound-checking*, y finalmente se convierte (utilizando el método **as()**) el array de int a uno de double, imprimiendo el resultado. También se puede observar en estos pasos algunas funciones de formateo de los *streams* de la STL.

Especialización

Una característica muy conveniente de las *templates* de C++, es la especialización. Esto permite definir un comportamiento específico para un tipo de datos particular de una *template*.

Son varias las veces donde es posible proveer una implementación más eficiente (o simplemente más *correcta*) para un dato en particular de lo que puede proveer un algoritmo genérico. Un ejemplo de esto puede ser un array de bool (que puede ser implementado utilizando bits). Incluso nuestra clase `Array` se puede beneficiar mucho de la especialización. ¿Que pasa si hiciera una copia del array `args`, que almacena punteros a `char`? ¿Que pasa si se liberaran esos punteros? La copia quedaría apuntando a memoria liberada, un error común.

Para evitar esto, se podría hacer una versión del `Array` especializada, para que en el constructor aloque memoria y en el destructor la libere. En realidad con nuestro ejemplo esto no puede hacerse por el parámetro por omisión, pero si no tuvieramos el segundo parámetro (Tamaño), podríamos especificar la *template* para que tenga una implementación particular para `char*`, de esta manera:

```
template <>
struct Array< char* >
{
    // Tipos y constantes: para facilitar la programación genérica
    typedef char* value_type;
    ////////// más miembros pero especificados con char* //////////
    // Array estático
```

```
    char* _array[Tamano];  
};
```

Una desventaja, es que hay que redeclarar la clase completa (incluso ambas clases podrían no tener los mismos métodos; recordar que cada **instancia** de una *template* es una clase **distinta**). Pero hay otra desventaja aun mayor en este caso. ¿Qué pasa si quisiéramos ahora utilizar un `Array< int* >`? ¿Habría que hacer toda la especialización de nuevo para `int*`? Eso sería muy engorroso e iría en contra de otro de los objetivos de las *templates*: reutilización de código. Para esto es que existe la **especialización parcial**. La idea detrás de este concepto es poder especializar una clase para punteros, pero no un puntero en particular, sino cualquier puntero. Para hacer esto podríamos declarar:

```
template < typename Tipo >  
struct Array< Tipo* >  
{  
    // Tipos y constantes: para facilitar la programación genérica  
    typedef Tipo* value_type;  
    ////////// más miembros pero especificados con Tipo* //////////  
    // Array estático  
    Tipo* _array[Tamano];  
};
```

De esta manera, al crear un objeto del tipo `Array< int* >` se instanciaría una sola *template*, que se compartiría con otras instancias que utilicen punteros, como `Array< double* >`.

No vamos a entrar más en detalle con respecto a especialización porque es un tema complejo que excede el objetivo de esta clase. Siempre pueden ampliar en internet o en la bibliografía recomendada por la cátedra.

STL

STL significa **Standard Template Library** (en castellano Biblioteca Estándar de Plantillas) y es parte de la biblioteca estándar de C++ (en conjunto con las funciones de la biblioteca estándar de C). El objetivo principal de dicha biblioteca es albergar algoritmos y tipos de datos (en particular lo que se conoce como contenedores) tan básicos que prácticamente todo programa no trivial haga uso de al menos alguno de ellos.

A diferencia de otras bibliotecas estándar (como la de Java, Python o .NET), no pretende tener respuesta a todas las necesidades que alguna vez puede tener un programador en su vida, sólo busca crear una base, un mínimo común múltiplo, que sirva para construir bibliotecas o programas más complejos y específicos. En este sentido es muy similar a la biblioteca estándar de C, sólo que la STL va un paso más allá, implementando estructuras de datos básicas (listas, árboles, heaps, vectores dinámicos, etc.). Por lo tanto uno de los grandes objetivos de diseño de la STL es la genericidad. La STL siempre evitará tomar decisiones por el usuario (o al menos siempre habrá forma de cambiar el comportamiento por omisión).

Otra característica es la eficiencia. C++ pretende ser un lenguaje apto para escribir programas con grandes requerimientos de performance (sistemas de tiempo real por ejemplo). Es por esto que la STL sacrifica facilidad de uso por performance. Si bien no es extremadamente compleja de usar, hay construcciones que pareciera que podrían ser más simples y muchas veces no lo son para poder dar lugar a optimizaciones (un ejemplo clásico es que los métodos tipo **pop()** no retornan el elemento eliminado, hay que usar el método **top()** para obtener el tope y luego hacer el **pop()** para *sacarlo*). Otra particularidad en este sentido es que están especificado hasta los ordenes de complejidad de los algoritmos de la STL, permitiendo tener un control bastante fino sobre la performance incluso sin conocer la implementación.

Namespace std

Todas las clases de la STL están disponibles dentro del namespace **std**. Para poder encontrar una clase se debe mencionar explícitamente el namespace, por ejemplo:

```
#include <list>
...
std::list<int> l;
```

A menos que necesite utilizarlo a lo largo del archivo y prefiera ahorrar el trabajo de escribirlo cada vez. En ese caso la palabra clave **using** me permite declarar que lo estaré usando:

```
#include <iostream>
using namespace std;
...
cout << "Hola" << endl;
```

String

La clase string es la más común de encontrar reimplementado en muchas librerías. En algunos casos son librerías creadas antes del standard; en otros casos los autores de las librerías no les gustaba el standard.

La clase string está definida en la cabecera **string**, sin .h, (al igual que el resto de la STL)

```
#include<string>

std::string s = "Hola";
```

La clase string tiene todo lo que esperaríamos de ella:

- Constructor de copia
- Operadores =, <, >, ==, []
- Operador + para concatenar
- Operador = de un `char*` (como en el ejemplo arriba)
- Un método `c_str()` que devuelve una referencia a un string formato C (con el `\0` al final)

Contenedores

Estos son los contenedores actualmente disponibles:

vector	Vector unidimensional de T
list	Lista doblemente enlazada de T
deque	Cola con doble entrada de T
queue	Cola de T
stack	Pila de T
map	Array asociativo de T con K por clave
set	Conjunto de T
bitset	Array de bits o booleans

Adicionalmente existen **multimap**, **multiset** y **priority_queue**. Todos los contenedores tienen similares modos de acceso, pero dependiendo de su tipo tendrán distinto tiempo de acceso. Por ejemplo:

- El contenido **list** tiene tiempo de acceso constante para acceder al primer y ultimo elemento, así como para insertar en el medio. Sin embargo para acceder a una posición random el tiempo de acceso es $O(n)$.
- Por otro lado **vector** tiene acceso random en tiempo constante, sin embargo para insertar un elemento el tiempo sera $O(n)$

Se almacenan siempre copias

Todos los contenedores de la STL almacenan copias de los objetos cargados. Es decir:

```
std::list<std::string> s;  
s.push_back(std::string("Hola Mundo"));
```

Crear primero un objeto string a partir de el "Hola Mundo" (formato C) en el code segment, y a continuación este string sera copiado a un nuevo string almacenado en la lista.
Si lo que deseamos es que no se hagan copias debemos recurrir a usar punteros:

```
std::list<std::string*> s;
```

Debido a esto es importante que al usar polimorfismo no intentemos hacer cosas como:

```
std::list<Figura> figs;  
figs.push_back(Triangulo(10));
```

Salvo que **Figura** sea abstracta o no tenga constructor de copia el código anterior funcionara pero el objeto almacenado en la lista no sera un **Triangulo**;

No usar polimórficamente

Ningún contenedor de la STL tiene destructor virtual. Por lo tanto usarlo polimórficamente puede conducir a errores de memoria.

Acceso a los contenedores básicos

Aquí hay una breve lista de los métodos que tienen los contenedores básicos. Por favor referirse a un libro para ver la lista completa:

stack:

```
bool empty()           //Indica si esta vacía  
void push(T &t)        //coloca un elemento arriba  
void pop()             //quita el primer elemento  
T& top()               //devuelve una referencia al primer elemento
```

queue:

```
bool empty()           //Indica si esta vacía  
void push(T &t)        //coloca un elemento al final  
void pop()             //quita el primer elemento  
T& front()             //devuelve una referencia al primer elemento
```

list:

```
bool empty()           //Indica si esta vacía  
void push_front(const T&) //coloca un elemento al principio  
void push_back(const T&) //coloca un elemento al final  
void pop_front()       //quita el primer elemento
```

```
void pop_back()           //quita el ultimo elemento
T& front()                //devuelve una referencia al primer
    elemento
T& back()                  //devuelve una referencia al último
    elemento
```

Iteradores

Todos los contenedores pueden ser accedidos por iteradores. Estos son como punteros para recorrer sus miembros. La forma mas común de usarlos es como se muestra a continuación:

```
std::list<int> lista;
std::list<int>::iterator i;
for(i=lista.begin(); i!=lista.end(); ++i)
    std::cout << *i;
```

Los iteradores tienen los operadores ++, * y -> para comportarse como punteros.

Iteradores const

En ocasiones tenemos contenedores que tienen el indicador de const colocado. En estos casos no podemos obtener un iterador de ellos ya que violaríamos la restricción de const y nos da un error de compilación.

Para estos casos los contenedores tienen también iteradores const que no permiten que se modifique el contenedor. Su uso es muy similar:

```
const std::list<int> lista;
std::list<int>::const_iterator i;
for(i=lista.begin(); i!=lista.end(); ++i)
    std::cout << *i;
```

Naturalmente el operador * en lugar de devolver una referencia devuelve una referencia **const**.

Contenedor asociativo

El contenedor asociativo map permite almacenar objetos usando otros objetos por clave. Internamente implementa esto en un árbol balanceado y la performance de acceso es $O(\log(n))$. Para declarar un map se deben indicar los 2 tipos, el de la clave y el del dato:

```
std::map<std::string, std::string> colores;
colores["banana"] = "amarillo";
colores["manzana"] = "rojo";
```

En el caso del ejemplo estoy usando **string** como tipo para la clave y para el dato, pero no tienen por que ser iguales. Los datos para almacenar no tienen por que ser de la STL, pero de no serlo hay requisitos adicionales. El tipo que se este usando como clave necesita tener el `operator ==` y el `operator <` correctamente definidos, ya que con estos implementara el árbol. Adicionalmente, y como en el resto de los contenedores, es necesario un constructor de copia para ambos.

Algoritmos

La STL incorpora varios algoritmos, sin embargo su uso no es simple. Referirse a un libro para detalles.

Para dar un ejemplo existe una funcion sort que recibe como parametros dos iteradores. Por ejemplo para una lista seria:

```
std::vector v;  
...  
std::sort(v.begin(), v.end());
```

Otro ejemplo es la funcion find que tambien toma dos iteradores y un objeto a buscar:

```
nums_iter = std::find(list_nums.begin(), list_nums.end(), 3);
```

Estas funciones tienen requerimientos para los parametros que reciben. Referirse a la documentación para detalles.

Input Output

[PENDIENTE]

Por favor referirse a la bibliografía existente. Algunas referencias:

```
iostream -> http://www.cplusplus.com/reference/iostream/  
STL -> http://www.sgi.com/tech/stl/table\_of\_contents.html
```