Los algoritmos de la STL

par Rodrigo Pons (home)

Date de publication : 06/11/2008

Dernière mise à jour : 05/07/2010

Este artículo es una descripción de los conceptos necesarios para una buena utilización de los algoritmos de STL.
Proporciona informaciones sobre cada uno de estos algoritmos (complejidad, precisiones sobre la utilización, etc), y un ejemplo de utilización para cada uno de ellos.

volver a la página inicial

Introducción

La librería "*algorithm*" es una parte de la STL. Es definida en 2 cabeceras : <algorithm> y <numeric>.
Los algoritmos de la STL forman una herramienta extremadamente poderosa. En ciertas situaciones de la vida real de un programador C++, esta parte de la STL puede revelarse realmente indispensable. Más de una vez, he debido desarrollar pequeñas programas, simples y eficaces, y con obligaciones de cualidad y de tiempo que hicieron de este desarrollo un verdadero desafío. Y si siempre lo he logrado, ha sido a menudo gracias a estos algoritmos de los que voy hablarle aquí.
Este documento es una vision general sobre el conjunto de los algoritmos. Censé más de 50 funciones, y la mayoría de ellas tienen varias versiones sobrecargadas. Entonces, no puedo razonablemente hablar en detalle de cada una de ellas, tanto más que se reencuentra una filosofía común. Además, voy a hablar aquí sólo de los algoritmos que forman parte del estándar, y que por lo tanto, estarán disponibles en todos los buenos compiladores.
Para ilustrar ciertos algoritmos, escogí hacer un pequeño programa que usted encontrará en archivo adjunto al final de este artículo. Algunos no estan en este pequeño programa, pero estan ilustrados con unas líneas de código, en el capítulo dedicado a ellos.

I - Generalidades

I-A - Por qué utilizar los algoritmos de STL

Aparte de unos casos pocos frecuentes, es siempre preferible utilizar los algoritmos proporcionados por STL que algoritmos "hechos a mano".
Si eligi comenzar mi artículo con esta frase, no es anodino, porque puede ser sujeta a polémica. Sin embargo, muchos desarrolladores a menudo olvidan que un buen programa debe reunir varias cualidades, y que no basta que sea solamente más rápido, o más transportable, o más mantenible, o más modular, etc. La mayoría de las veces, debe ser un buen compromiso de todas estas cualidades. Y los algoritmos (y la STL más generalmente) proponen el mejor compromiso. ¿Por qué?

- La abstracción:
  Estos algoritmos ofrecen un nivel de abstracción que permite un desarrollo muy limpio. Estos algoritmos permiten aplicar una operación sobre un conjunto de elementos sin preocuparse en detalle de la naturaleza de estos elementos, y sin tener que hacer un código laborioso y especializado para hacerlo.
- La ganancia de tiempo, en término de "tiempo de desarrollo":
  Numerosos algoritmos ya son hechos y permiten resolver los problemas más frecuentes unas líneas de código.
- La robustez del código:
  Estos algoritmos son utilizados por millares de programas y respetan estándares que aseguran la robustez.
- La rapidez de ejecución:
  La utilización de los algoritmos de la STL suprime ciertas informaciones redundantes (prueba sobre el iterador de fin de bucle por ejemplo). Además, los desarrolladores que hacen estos algoritmos conocen las estructuras internas de los contenedores en los cuales trabajan y pueden así optimizar mucho mejor que prodriamos hacerlo. Finalmente, los algoritmos de la STL utilizan los métodos más recientes del "arte", pues los más eficaces.
- La mantenabilidad del código:
  La STL proporciona un estándar conocido y reconocido por millares de desarrolladores. Así, cuando un desarrollador se encuentra en frente de un código que utiliza estos algoritmos, está en terreno conocido y pasa menos tiempo para entender este código.

I-B - Functor, predicados...

Todos estos tipos de funciones y de functor (o funtor, o objeto-función, o funciones-objetos) están muy presentes en la STL, y es indispensable conocer un mínimo de lo que es para utilizar correctamente los algoritmos. Entonces, voy a comenzar por introducir un poco vocabulario. Es bastante desagradable lo convengo, pero estoy convencido que usted me agradecerá más tarde por eso.

I-B-1 - Functor

Ni el C ni el C++ permiten pasar funciones en parámetros. Es la razón por la cual hay que utilizar a punteros de función. Un functor hace la misma cosa que un puntero de función, pero permite hacer mas cosas.
Un functor es, sencillamente, una clase que sobrecarga al operador de llamada de función (*operator ()*). El functor presenta por lo menos 3 ventajas importantes comparado al puntero de función:

- Eficacia (rapidez de ejecución): su operador de llamada de función puede ser declarado (*inline*).
- La utilización de las variables miembros del functor permite una flexibilidad de utilización que es complicado de obtener con punteros de función.
- El functor puede tener un estado.
- El functor presenta un enfoque objeto que es apreciable para un desarollador acostumbrado a esto.

Capítulo de zator.com sobre los functores

I-B-2 - Predicado

Un predicado es una función que devuelve un booleano. Muchos algoritmos de la STL utilizan un predicado.
Capítulo de zator.com sobre los predicados

I-B-3 - Función pura

Una función pura es una función cuyo resultado depende únicamente de sus parámetros.

I-B-4 - Clase predicado

Una clase predicado es un functor cuyo el operador de llamada de función devulve un booleano.
Tengo que anotar que los algoritmos de la STL pueden efectuar copias "escondida" de una clase predicado. Es por eso que el operador de llamada de función de una clase predicado tiene que ser una función pura. En efecto, es difícil manejar perfectamente el contexto (en particular las variables miembros) de la clase copiada cuando la copia es más o menos escondida dentro de la STL.

I-C - ptr_fun(), mem_fun() et mem_fun_ref()

Estas funciones, llamadas función-adaptador, no son muy agradables a leer ni a escribir, y sus nombres no son muy explícitos. Sin embargo, tienen un papel muy importante. Son definidas en el cabecero <functional>. ¿Para qué sirven?
Una vez más, un poco de código vale más que un discurso largo. Fijaos el código siguiente:

```
// declaración de una función dum1 que toma una referencia hacia una instancia de UnaClase en parámetro.
void dum1(UnaClase & objet);

// clase UnaClase que posee una función miembro dum2
class UnaClase
{
public :
void dum2();
```

```
};

// un vector de UnaClase
std::vector<UnaClase> v1;

// un vector de punteros sobre UnaClase
std::vector<UnaClase*> v2;
```

Si se desea aplicar nuestra función *dum1()* a todo el vector *v1*, podemos hacer así:

```
for_each( v1.begin(), v1.end(), dum1 ) ; // vale
```

Este código funciona porque *dum1()* no es una función miembra.
En cambio, si se desea aplicar la función miembro *UnaClase::dum2()* en todo el vector, el código siguiente no funcionará:

```
for_each( v1.begin(), v1.end(), &UnaClase::dum2 ); // no vale
```

Para hacer simple, el convenio de la STL no permite esto.
Entonces, habrá que utilizar los adaptadores de función objeto, es decir:

- *mem_fun_ref()* en caso de que el contenedor contiene instancias del objeto (es el caso de *v1* en mi ejemplo).
- *mem_fun()* en caso de que el contenedor contiene punteros sobre instancias de objeto (es el caso de *v2* en mi ejemplo).

```
std::for_each(v1.begin(), v1.end(), mem_fun_ref( &UnaClase::dum2 ) ); //vale
std::for_each( v2.begin(), v2.end(), mem_fun( &UnaClase::dum2 ) ); //vale
```

En lo que se refiere a *ptr_fun()*, vamos a necesitarlo para utilizar los 4 adaptadores de función estándares de la STL, es decir *not1()*, *not2()*, *bind1st()* y *bind2nd()*. En efecto, estos adaptadores necesitan unos *typedefs* específicos que son declarados en *ptr_fun()*.
Por ejemplo, tomemos el caso de un predicado que nos dice si una instancia de *UnaClase* es válida:

```
bool IsValid(const UnaClase* object);
```

Si quiero encontrar el primer objeto de *v2* que no es válido, sería intentado hacer:

```
std::vector<UnaClase*>::iterator i = std::find_if( v2.begin(),
 v2.end(), std::not1( IsValid ) ); //no vale
```

El problema es que esto no compilará. Habrá que utilizar un adaptador:

```
std::vector<UnaClase*>::iterator i = std::find_if( v2.begin(),
 v2.end(), std::not1( std::ptr_fun( IsValid ) ) ); //ok
```

De hecho, los adaptadores de funciones-objeto (*mem_fun*, *mem_fun_ref* y *ptr_fun*) sólo definen tipos los que las funciones objeto necesitan. Si quiere saber más sobre este tema, le invito a echar una ojo al item 41 del indispensable **"Efective STL"** de Scott Meyers.

Referencias:
pagina sgi sobre la STL
functional members de la STL para VS8 en la MSDN2

I-D - bind1st() y bind2nd()

En líneas generales, estas 2 funciones permiten transformar un predicado que toma 2 argumentos en un predicado que toma sólo uno, dándole el valor del argumento faltante.

Un poco de código para comprender mejor:

```
// comenzamos por definir un functor que va permitir a nosotros construir un contenedor rápido
struct Identity
{
 Identity():count_(0){}
 int operator () () { return count_++; }
 int count_;
};

// también definimos un predicado binario que compara 2 enteros
bool IsSup(int a, int b) {return a>b;}

// declaramos un vector de enteros de 5 elementos
std::vector<int> v(5);

// así como un iterador que va permitir a nosotros trabajar en nuestro vector
std::vector<int>::iterator it ;

// y finalmente, rellenado nuestro vector gracias al algoritmo generate() y nuestro functor
std::generate( v.begin(), v.end(), Identity() );
// ahora tenemos:  v = {0,1,2,3,4}
```

*Ver la descripción de la función **generate()***

Ahora, querríamos efectuar una búsqueda para encontrar el primer elemento de nuestro vector que sea superior a 2. El algoritmo ***find_if()*** es ideal para hacer esto.
Nos gustaría poder escribir algo así:

```
it = std::find_if( v.begin(), v.end(), IsSup(2) ); //no vale
```

Por supuesto, esto no compila. *IsSup* es un predicado binario, entonces necesita 2 argumentos. La solución es pasar por un *binder*:

```
it = std::find_if( v.begin(), v.end(), std::bind2nd( std::ptr_fun( IsSup ) , 2 ) ); //vale
// el resultado de esta línea es que it apunta sobre 3 (el primer valor superior a 2)
```

*Ver la descripción de la función **find_if()***

Vamos a ver mas en detalle lo que esta última línea de código efectúa, porque en una línea, se hacen muchas cosas. Así, como explicado en **capítulo precedente**, *ptr_fun()* es necesario para que el adaptador *bind2nd()* pueda funcionar, no diré de eso más aquí.
find_if() va a crear un iterador escondido con el cual va a recorrer el contenedor segun los límites que le serán pasados en argumentos. Va a aplicar un predicado pasándole el contenido de este iterador escondido hasta que este predicado devuelva *"true"* o que el límite superior sea sobrepasado. En nuestro ejemplo, *v.begin()* es el límite inferior, *v.end()* el límite superior, y *IsSup* nuestro predicado.
El problema es que *find_if()* toma un predicado unario, ahora si *IsSup()* está bien un predicado, es un predicado binario. Y está alla dónde *bind2nd()* interviene: va a transformar (adaptar) nuestro predicado binario en un predicado unario. De hecho, es como si crearía una nueva función *IsSup* que se parecería a esto:

```
Bool IsSup( std::vector<int>::iterator it ){ return (*it)>2; }
```

bind2nd() suprime el secundo argumento de *IsSup* y reemplaza, en el cuerpo de la función, este argumento por el valor que pasamos en argumento a *bind2nd()*.
bind1st() hace exactamente la misma cosa pero él reemplaza el primero argumento. Por ejemplo, si se reemplaza, en la línea que se acaba de analizar, *bind2nd* por *bind1st*:

```
IntIt it = std::find_if( v.begin(), v.end(), std::bind1st( std::ptr_fun( IsSup ), 2 ) );
//resultado: it apunta sobre 0, ya que es el primer elemento del array que sea inferior a 2.
```

En la literatura, encontrará diferentes términos utilizados para nombrar estas funciones particulares que son *bind1st()* y *bind2nd()*. En efecto, podremos reencontrarlos con el nombre de *adaptador de función*, *binder*, o todavía *reductor de functor*.

Las combinaciones de adaptadores y de reductores pueden dar el código bastante indigesto. Así otras soluciones son propuestas por boost. Ver particularmente **boost::bind** y **boost::function**.

A ver tambien:
FAQ DVP sobre los reductores

I-E - A propósito de los iteradores de inserción

Un iterador de inserción (*inserter* en inglés) es un iterador un poco particular. Un iterador clásico se contenta de recorrer un contenedor. Un inserter va a añadir, en otro contenedor, el elemento sobre el cual despunta .
Los iteradores de inserción son muy útiles para algoritmos de copia de elementos, porque ellos permiten construir un nuevo contenedor sin tener que hacer un *new*.
Existen varias tipos de iteradores de inserción.

Los más comunes son:

- back_inserter: añade el elemento al fin del nuevo contenedor. *(ver función **merge()** para un ejemplo)*
- front_insert: añade el elemento al principio del nuevo contenedor
- ostream_iterator: añade el elemento en un flujo de tipo ostream

Usted encontrará ejemplos de utilización de los iteradores de inserción en este documento, en particular en **el capítulo sobre la función copy()**.
Referencias:
FAQ C++ sobre los iteradores de inserción
La documentación de Rogueware sobre los iteradores de inserción
La documentación de Zator sobre los iteradores de inserción

I-F - A propósito de la eficiencia.

Con el fin de verificar ciertos aspectos relacionados al eficiencia de la Librería "*algorithm*", efectué varias series de pruebas. Estas pruebas no son detalladas aquí porque el objetivo de este artículo no es hacer una comparación de eficiencia entre diversos modos de programar en C++.
Para estas pruebas, utilicé la función **generate ()**. Escogí esta función para varias razones. En primer lugar, es una función simple de utilización y que necesita pocos líneas de código. Luego, es una función modificante, lo que nos permite de verificar fácilmente su buen funcionamiento. Y finalmente, porque es una función que es fácil para imitar con un boucle *for* , lo que devuelve la comparación más aigüe.
Para los resultados, utilicé dos métodos de cuantificación:

- El analizador de eficiencia *VTune* (cuando escribo estas líneas, una versión de evaluación de 30 días está disponible en el sitio de Intel).
- Un cronómetro "hecho mano" que utiliza las funciones de programa "alto eficiencia" de Windows (*QueryPerformanceCounter()*, etc).

I-E-1 - Descripción de las pruebas

Para estas pruebas, he escrito 3 funciones *gen1()*, *gen2()* y *gen3()*. Todas estas funciones hacen exactamente la misma cosa: ellas rellenan un contenedor para lo que *contenedor[i] = i*. Sin embargo, cada función es hecha de manera diferente.
El contenedor (*container*) y el tamaño de este contenedor (*c_size*) son variables globales, con el fin de aliviar el código de las funciones. También utilizo un iterador *MyIterator* que será definido con arreglo al tipo del contenedor.

He aquí el código de estas tres funciones:

```
// la primera función gen1() que efectúa un boucle for simple y utiliza el iterador correspondiente al contenedor
void gen1()
{
 MyIterator end = container.end();
 int i = -1;
 for (MyIterator it = container.begin(); it != end; ++it)
  *it = ++i;
}
```

```
// la segunda función gen2() que utiliza a un puntero para recorrer
// el contenedor. Atención, este método funciona sólo si todos los elementos del
// contenedor utilizado son asegurados de ser adyacentes en memoria.
void gen2()
{
 int i = -1;
 int * end = &container[c_size-1];
 for(int * it = &container[0]-1; it != end;)
  *++it = ++i;
}
```

```
// la tercera función gen3() que utiliza el algoritmo generate() y una functor MyGenerate
struct MyGenerate
{
 explicit MyGenerate() : count_(0) {}
 int operator () () { return ++count_; }

private:
 int count_;
};

void gen3()
{
 std::generate( container.begin(), container.end(), MyGenerate() );
}
```

I-E-2 - Resultados

Resultados obtenidos con:

```
std::vector<int> container;
const int c_size = 5000000;
typedef std::vector<int>::iterator MyIterator;
```

gen3() es más rápido, por término medio, que *gen1()* del 4,8 %. No es enorme, pero ya vemos que la utilización de un algoritmo es más eficaz que de utilizar iterador.
gen3() es más rápido, por término medio, que *gen2()* del 0,2 %, que no es representativo verdaderamente, pero el operador [] es el modo más rápido de acceder a un elemento de uno *vector*, y el bucle de *gen2()* es muy optimizado.

Resultados obtenidos con:

```
std::deque<int> container;
const int c_size = 500000;
typedef std::deque<int>::iterator MyIterator;
```

El contenedor *deque* no almacena sus elementos de manera adyacente, entonces no podemos utilizar la función *gen2()* para esta prueba.
gen3() es más rápido, por término medio, que *gen1()* del 6,5 %. La diferencia es más importante que con un *vector* porque la memoria de una *deque* es administrada de modo un poco particular, y el algoritmo *generate()* "sabe " cómo optimizar su recorrido del contenedor. Vemos pues que, desde el punto de vista del eficienca, los algoritmos son por lo menos tanto rápidos como funciones muy bien optimizadas.

I-E-3 - Eficiencia y functor

Los functores permiten aumentar la rapidez de ejecución, en comparación con un puntero de función, porque se puede poner *inline* el operador de llamada de función. Sin embargo, hay que tener cuidado de una cosa, es que el functor esta copiado frecuentemente, de modo escondido, cuando se utiliza los algoritmos de STL. Entonces es importante asegurarse que el functor esta el más ligero posible.
Usted encontrará un ejemplo de esto en el código proporcionado con este artículo, en el functor Synchro. Es exactamente lo que no hay que hacer: un functor muy pesado.
En caso de functor en el cual seríamos intentados añadir código, puede ser interesante utilizar la astucia siguiente: crear functors específicos que puntan sobre una instancia de este grande functor, y que será llamado por los algoritmos que lo necesitan, estos algoritmos que no utilizan directamente todavia al grande functor.
Por ejemplo, en el código proporcionado con este artículo, encontramos el functor siguiente:

```
struct Synchro
{
 Synchro( const StringVector & artists )
  : artists_( artists )
 {
  std::vector<AlbumList>( artists_.size() ).swap( albums_ );
  StringPairs().swap( result_ );
  current_ = albums_.begin();
 }

 void operator () (const std::string & str)
 {
  AlbumManager::FindAllAlbumsOf( str, *current_ );
  std::sort( (*current_).begin(), (*current_).end() );
  std::for_each( albums_.begin(), current_, Synchro::Compare );
  current_++;
 }

 static void GetResult(StringPairs & result){ result = result_; }

private:
 std::vector<AlbumList> albums_;
 const StringVector & artists_;
 static std::vector<AlbumList>::iterator current_;
 static StringPairs result_;

 static void Compare(const AlbumList & list)
 {
  if ( current_->size() == list.size()
    && std::equal( current_->begin(),
      current_->end(),
      list.begin(),
      std::mem_fun_ref( &Album::ReleasedSameYear ) ) )
   {
    result_.push_back( StringPair( (*current_)[0].GetArtist().GetName(),
 list[0].GetArtist().GetName() ) );
   }
 }
};
```

Este código debe ciertamente parecerle bastante oscuro, porque esta sacado de su contexto y no comentado. Pero el objetivo aquí no es de comprenderlo, sino justo ver que tenemos aquí un grande functor con muchos objetos y funciones miembros, y que este functor va a ser copiado frecuentemente de modo escondido, haciéndonos perder mucho en término de eficiencia.
Con el fin de evitar esto, es preferible añadir un functor intermediario que se encargará de llamar al operador de llamada de función del grande functor:

```
class SynchroCompare
{
public:
 SynchroCompare(Synchro* synchro) : synchro_(synchro){}
```

```
 void operator () (const std::string & str)
 {
  synchro_->operator()(str);
 }

private:
 Synchro* synchro_;
};
```

volver al principio

II - Lista de los algoritmos

Para cada uno de los algoritmos, voy a proporcionar un ejemplo de utilización. Usted encontrará dos tipos de código:

1. Código extraído del programa AlbumManager que usted encontrará al fin de este artículo.
Para estos extractos, utilizo una clase Album del que he aquí la versión simplificada:

```
class Album
{
public:
 // constructores, "getter" y otros
private:
 Artist   artist_; // un objeto de tipo Artist
 std::string  title_; // titulo del álbum
 unsigned short year_; // fecha de salida del álbum
 unsigned short nbTracks_; // número de pistas en el álbum
};
```

En ciertos ejemplos, definiré unas funcionalidades suplementarias para esta clase.
En estos extractos de código, utilizaré regularmente 2 vectores:
*main_list_* y *second_list_* que son 2 vectores de Álbum.

2. Código escrito directamente en el capítulo correspondiente. En este caso, utilizaré a uno *std::vector* que declararé como un array C de esta manera:

```
std::vector<int> v = {1,2,3,4};
```

Este código no compila, pero en un objetivo evidente de claridad, utilizo esta notación en lugar de:

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
```

Que, es correcta y compila, pero demasiado verboso.

II-A - Algoritmos no modificantes

II-A-1 - for_each()

Complejidad: lineal.
Efectúa una operación sobre cada elemento de un intervalo.
Extraído de AlbumManager:

```
// Sobrecarga del operador de indireccion
std::ostream & operator <<(std::ostream & stream, const Album & album)
```

```
{
 stream << album.title_ << ", " << album.artist_.GetName() << ", " << album.year_;
 stream << ", nb tracks: " << album.nbTracks_ << std::endl;
 return stream;
}

// Fija un álbum en la salida estándar utilizando el operador de indireccion definido encima
void PrintAlbum(const Album & album)
{
 std::cout << album;
}

// Fija una lista de álbums en la salida estándar
void PrintAlbumList(std::vector<Album> albums)
{
 std::for_each( albums.begin(), albums.end(), PrintAlbum );
}
```

II-A-2 - find() y find_if()

Complejidad: lineal

*find()*devuelve el primer elemento de un intervalo que es igual (operador ==) a un valor dado.
Extraído de AlbumManager:

```
std::string artistName = "un artista" ;

// tmpStrVect_ es un vector de string que contiene nombres de artistas
// El código siguiente añade a un nuevo artista en tmpStrVect_ si este artista no está ya presente en este vector
if ( std::find(tmpStrVect_.begin(), tmpStrVect_.end(), artistName ) == tmpStrVect_.end() )
{
 tmpStrVect_.push_back( artistName );
}
```

find_if() hago la misma cosa pero utiliza un predicado en lugar del operador ==.
Extraído de AlbumManager:

```
// HaveSameTitle ss un predicado que devuelve true si el título pasado en parámetro es lo mismo que el álbum
bool Album::HaveSameTitle(std::string title) const { return title_ == title; }

// Esta función recobra un álbum con arreglo a su título
bool FindByTitle(const std::string & title, Album & album)
{
 AlbumList::iterator searched =
  std::find_if( main_list_.begin(),
    main_list_.end(),
    std::bind2nd ( std::mem_fun_ref<bool, Album, std::string>(&Album::HaveSameTitle), title) );

 if ( searched == main_list_.end() )
  return false;

 album = *searched;
 return true;
}
```

II-A-3 - find_end()

Complejidad: generalmente cuadrático. Sin embargo, si los iteradores ambos son bidireccionales, entonces complejidad medio es lineal.
find_end() parécese mucho a *search()*. La diferencia es que *search()* busca el primer intervalo, mientras que *find_end()* busca el último intervalo.

```
std::vector<int> v1 = {2, 6, 7, 8, 4, 6, 7, 8, 3} ; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {6, 7, 8}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator result = std::find_end( v1.begin(), v1.end(), v2.begin(), v2.end() );
```

```
//Devuelve un iterateur apuntando sobre el 6° elemento de v1
```

II-A-4 - find_first_of()

Complejidad: O(N.M), N y M son el tamaño de cada uno de los intervalos.
Busca el primer elemento *e1* de un intervalo *I1* como *e1* sea igual a cualquier elemento de un otro intervalo *I2*.
Extraído de AlbumManager:

```
// Esta función construye un vector con los álbumes que son comunes de dos vectores de álbumes
bool AlbumManager::GetCommonAlbums(std::vector<Album> & albums)
{
 // empezamos por vaciar el vector el resultado
 std::vector<Album>().swap( albums );

 // declaramos varios iteradores que necesitamos en el bucle
 std::vector<Album>::iterator begin = main_list_.begin();
 std::vector<Album>::iterator end = main_list_.end();
 std::vector<Album>::iterator it;

 // A cada iteración, buscamos el primer álbum común de las 2 listas, luego, si se hemos encontrado uno,
 // resituamos begin con el fin de empezar de nuevo el bucle ignorando la parte de main_list_ que
 // ya ha sido tratada.
 do
 {
  it = std::find_first_of( begin, main_list_.end(), second_list_.begin(), second_list_.end() );
  if ( it != end )
  {
   albums.push_back( *it );
   begin = it + 1;
  }
 }
 while ( it != end ); // si it == end, esto significa que se encontró nada, entonces podemos parar

 return ( albums.size() == 0 ) ? false : true;
}
```

II-A-5 - adjacent_find()

Complejidad: lineal.
Busca dos elementos adyacentes que son idénticos (por un criterio).
Extraído de AlbumManager:

```
// búsqueda de doblones en un vector.
// efectuamos un bucle sobre el vector fusionado, y cada vez
// que que encontramos un doblón, lo añadimos al vector resultado (result)
std::vector<Album>::iterator found = mergedList.begin();
do
{
 found = std::adjacent_find( found, mergedList.end() );
 if ( found != mergedList.end() )
 {
  result.push_back( *found );
  found++;
 }
} while ( found != mergedList.end() );
```

II-A-6 - count() y count_if()

Complejidad: lineal
count() devuelve el número de elementos que son iguales (operador ==) a un elemento dado.

```
Std ::vector <int> v = {2,3,4,2,4,2} ; // no compila (cf. capítulo 2.0)
int count = std::count(v.begin(), v.end(), 2) ; //devuelve 3
```

count_if() hace la misma cosa, pero utiliza un predicado dado al sitio del operador ==.
Extraído de AlbumManager:

```
bool Album::HaveSameArtist(std::string artist) const { return artist_.GetName() == artist; }

std::string artistName = "un artista";
int nbAlbums = (int) std::count_if( main_list_.begin(),
  cut,
  std::bind2nd( std::mem_fun_ref<bool, Album, std::string>( &Album::HaveSameArtist ), artistName) );
```

II-A-7 - mismatch()

Complejidad: lineal.
Devuelve los primeros elementos, de dos intervalos, que difieren.

⚠ *Ambos intervalos comparados deben tener el mismo número de elementos. Si no, el resultado es indeterminado.*

```
std::vector<int> v1 = {3,4,5,8}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {3,4,6,8}; // no compila (cf. capítulo 2.0)
// v1.size() = v2.size()  -> vale

typedef std::vector<int>::iterator IntIt;
std::pair<IntIt, IntIt> result = std::mismatch( v1.begin(), v1.end(), v2.begin() );
//resultado: *(result.first) = 5, *(result.second) = 6
```

II-A-8 - equal()

Complejidad: lineal.
Compara dos intervalos, elemento por elemento, y devuelve *true* si estos dos intervalos son idénticos.

⚠ *Ambos intervalos comparados deben tener el mismo número de elementos. Si no, el resultado es indeterminado.*

Usted encontrará un ejemplo concreto de utilización de *equal()* en el código de AlbumManager, pero él es utilizado en un lugar un poco complicado, entonces prefiero darle un ejemplo simple aquí:

```
std::vector<int> v1 = {1,2,3}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {1,2,3}; // no compila (cf. capítulo 2.0)
std::vector<int> v3 = {1,2,4}; // no compila (cf. capítulo 2.0)
bool b = std::equal(v1.begin(), v1.end(), v2.begin() ); //résultado: b = true
b = std::equal(v1.begin(), v1.end(), v3.begin() ); //résultado: b = false
```

II-A-9 - search() y search_n()

Complejidad: en el peor de los casos, ella es cuadratica, pero este caso es raro. La complejidad media es lineal.
search() Busca el primer caso de un intervalo en otro intervalo.

```
std::vector<int> v1 = {2, 3, 4, 6, 7, 8}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {6, 7, 8}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator result = std::search( v1.begin(), v1.end(), v2.begin(), v2.end() );
// devuelve un iterador que despunta sobre el 4° elemento de v1
```

Complejidad: lineal
search_n() busca los primeros *n* elementos consecutivos que son iguales a un elemento dado.

```
std::vector<int> v = {2,3,4,4,4,2}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator result = std::search_n( v.begin(), v.end(), 3, 4);
```

```
// devuelve un iterador que despunta sobre el 3° elemento de v
```

retour au debut

II-B - Algoritmos modificantes

II-B-1 - copy() y copy_backward()

Complejidad: lineal.
copy() copia un intervalo a partir del primer elemento.
Usted encontrará un ejemplo concreto de utilización de *copy()* en el código de AlbumManager, pero él es utilizado en un lugar un poco complicado, entonces prefiero darle un ejemplo simple aquí:

```
std::vector<int> v = {0,1,2,3,4}; // no compila (cf. capítulo 2.0)
std::copy( v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
// fija "0 1 2 3 4" en la salida estándar
```

copy_backward() hace la misma cosa que *copy()* excepto que él rellena el intervalo de destinación empezando por el fin:

```
std::vector<int> v2( v1.size());
std::copy_backward( v1.begin(), v1.end(), v2.end() );
// résultado: v2 = {0,1,2,3,4};
```

II-B-2 - swap() y swap_ranges()

Complejidad: constante.
swap() intercambia los elementos de dos contenedores.

```
std::vector<int> v1 = {1,2,3,7}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {4,5,6}; // no compila (cf. capítulo 2.0)
std::swap( v1, v2 );
// resultado: v1 = {4,5,6}, v2 = {1,2,3,7}
```

Complejidad: lineal.
swap_range() intercambia los elementos de dos contenedores.

Ambos intervalos comparados deben tener el mismo número de elementos. Si no, el resultado es indeterminado.

```
std::vector<int> v1 = {1,2,3,7}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {4,5,6}; // no compila (cf. capítulo 2.0)
std::swap_ranges( v1.begin(), v1.begin() + v2.size(), v2.begin() );
// resultado: v1 = {4,5,6,7}, v2 = {1,2,3}
```

Observamos en el ejemplo de arriba que swap_ranges() es aplicado únicamente sobre los tres primeros elementos de v1. Si, en el lugar de v1.begin() + v2.size() habíamos eligido v1.end(), el resultado habría sido indeterminado porque que v1 es más grande que v2.

II-B-3 - transform()

Complejidad: lineal.
Modifica elementos de un intervalo en un otro intervalo aplicando una función. Fijale que el intervalo de destinación puede ser el mismo que el intervalo fuente.

Ambos intervalos (fuente y destinación) deben tener el mismo número de elementos. Si no, el resultado es indeterminado.

Extraído de AlbumManager:

```
// "getter" que recupera el número de pista sobre un álbum
unsigned short GetNbTracks() const { return nbTracks_;}

// creamos un vector que va a contener sólo el número de pistas
std::vector<unsigned short> tracks( main_list_.size() );
std::transform( main_list_.begin(), main_list_.end(),
 tracks.begin(), std::mem_fun_ref( &Album::GetNbTracks ) );
```

II-B-4 - Algoritmos de sustitución

Complejidad: lineal.

II-B-4-1 - replace()

replace() reemplaza elementos que son iguales (operador ==) a un valor dado por otro valor.

```
std::vector<int> v = {1,4,3,4,1}; // no compila (cf. capítulo 2.0)
std::replace( v.begin(), v.end(), 4, 2);
// resultado: v = {1,2,3,2,1}
```

II-B-4-2 - replace_if()

replace_if() hace la misma cosa que *replace()* pero él utiliza un predicado dado en lugar del operador ==.

```
// IsOdd es un predicado que determina si un número es impar
 bool IsOdd(int number){ return number&1;}

 std::vector<int> v = {1,4,3,4,1}; // no compila (cf. capítulo 2.0)
 std::replace_if( v.begin(), v.end(), IsOdd, 0); // reemplaza todos los números impares de v1 por 0
 // resultado: v = {0,4,0,4,0}
```

II-B-4-3 - replace_copy()

replace_copy() reemplaza los elementos que tienen un valor especial copiando el intervalo entero.

```
std::vector<int> v1 = {1,4,3,4,1}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::replace_copy( v1.begin(), v1.end(), v2.begin(), 1, 0);
// resultado: v2 = {0,4,3,4,0}
```

II-B-4-4 - replace_copy_if()

replace_copy_if() hace la misma cosa que *replace_copy()* pero él copia sólo los elementos que corresponden a un predicado dado.

```
// IsOdd es un predicado que determina si un número es impar
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {1,4,3,4,1}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::replace_copy_if( v1.begin(), v1.end(), v2.begin(), IsOdd, 0);
//copia v1 en v2 reemplazando todos los números impares de v1 por 0
```

```
//resultado : v2 = {0,4,0,4,0}
```

II-B-5 - fill() y fill_n()

Complejidad: lineal.
fill() reemplaza cada elemento de un intervalo por un valor dado.

```
std::vector<int> v = {1,2,3,7}; // no compila (cf. capítulo 2.0)
std::fill( v.begin(), v1.end(), 5 );
//resultado: v = {5,5,5,5};
```

fill_n() hace la misma cosa pero se aplica sólo a *n* elementos del intervalo.

```
std::vector<int> v = {1,2,3,7}; // no compila (cf. capítulo 2.0)
std::fill( v.begin(), 2, 5 );
//resultado: v = {5,5,3,7};
```

II-B-6 - generate() y generate_n()

Complejidad: lineal.
generate() reemplaza cada elemento de un intervalo por el resultado de una operación.

```
int count = 0;
int MyGenerate(){ return count++; }

std::vector<int> v(5);
std::generate( v.begin(), v.end(), MyGenerate );
//resultado: v = {0,1,2,3,4}
```

generate_n() hace la misma cosa que *generate()* pero se aplica sólo a *n* elementos del intervalo. n debe ser inferior al tamaño del intervalo.

```
int count = 0;
int MyGenerate(){ return count++; }

std::vector<int> v(5, 0); // aqui v = {0,0,0,0,0}
std::generate_n( v.begin(), 3, MyGenerate );
//resultado: v = {0,1,2,0,0}
```

II-B-7 - algoritmos de supresión

Complejidad: lineal.

II-B-7-1 - remove()

remove() suprime los elementos que tienen un valor dado.

```
std::vector<int> v1 = {1,2,3,4}; // no compila (cf. capítulo 2.0)

std::remove( v1.begin(), v1.end(), 3); //resultado: v1 = {1,2,4,4}; en effet:
std::copy( v1.begin(), v1.end(), std::ostream_iterator<int>(std::cout, " "));
// fijara 1 2 4 4
```

Es por eso que hay que utilizar el iterador enviado por la función *remove()*:

```
std::vector<int> v1 = {1,2,3,4}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator it = std::remove( v1.begin(), v1.end(), 3);
std::copy( v1.begin(), it, std::ostream_iterator<int>(std::cout, " "));
```

```
// fijara 1 2 4
```

Otra solución, por ejemplo, cuando el contenedor contiene punteros, consiste en utilizar la función ***partition()*** luego la función miembro de borrado del contenedor, generalmente *erase()*.

II-B-7-2 - remove_if()

remove_if() hace lo mismo que *remove()* pero borra solamente los elementos correspondientes a un predicado dado.

```
// IsOdd es un predicado que determina si un nombre es impar
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {1,2,3,4,5,6,7}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator it = std::remove_if( v1.begin(), v1.end(), IsOdd);
std::copy( v1.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// fijara: 1 3 5 7
```

II-B-7-3 - remove_copy()

remove_copy() copia los elementos que no corresponden a un valor dado.

```
std::vector<int> v1 = {1,2,3,4,3,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::vector<int>::iterator it = std::remove_copy( v1.begin(), v1.end(), v2.begin(), 3);
std::copy( v2.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// fijara: 1 2 4 6
```

II-B-7-4 - remove_copy_if()

remove_copy_if() copia los elementos que no corresponden a un predicado dado.

```
// IsOdd es un predicado que determina si un nombre es impar
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {1,2,3,4,3,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::vector<int>::iterator it = std::remove_copy_if( v1.begin(), v1.end(), v2.begin(), IsOdd);
std::copy( v2.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// fijara: 2 4 6
```

II-B-8 - unique() y unique_copy()

Complejidad: lineal.
unique(): Cada vez que un grupo de duplicados adyacente (elementos que son iguales a sus predecesores) esta en un intervalo [principio, fin) dado, *unique()* suprime todo los duplicados menos el primero. *unique()* devuelve un iterador it tal como el intervalo [principio, it) no contenga duplicados. Los elementos del intervalo [it, fin) son dereferenciables ya, pero los elementos sobre los cuales puntan son indeterminados.
Usted encontrará un ejemplo concreto de utilización de *unique()* en el código de AlbumManager, pero él es utilizado de manera un poco particular, entonces prefiero darle un ejemplo simple aquí:

```
std::vector<int> v = {3,1,1,2,4}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator it = std::unique( v.begin(), v.end() );
std::copy( v.begin(), it, std::ostream_iterator<int>(std::cout, " "));
// va afficher : 3 2 1 4
```

La implementación exacta de unique() depende de su versión de la librería algorithm. El código más arriba utiliza el STL de MS VC 8, y vemos que la orden de los elementos es modificada por unique(). El estándar especifica solamente que el intervalo situado antes

del iterador devuelto por unique() no contiene doblones, no da ninguna precisión sobre la orden de estos elementos.

unique_copy(): unique-copy hace lo mismo que *unique()*, pero por encima, esto copia el resultado en otro contenedor.

```
std::vector<int> v1 = {1,2,2,3,4,4,3,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2;
std::unique_copy(v1.begin(), v1.end(), std::back_inserter(v2) );
// resultado: v2 = {1,2,3,4,3,6}
```

II-B-9 - reverse() y reverse_copy()

Complejidad: lineal.
reverse() invierte la orden de los elementos de un intervalo.

```
std::vector<int> v = {1,2,3,4,5} ; // no compila (cf. capítulo 2.0)
std::reverse( v1.begin(), v1.end() );
// resultado : v = {5,4,3,2,1}
```

reverse_copy() copia los elementos de un intervalo fuente hacia un intervalo destino invirtiendo su orden.

⚠ *Ambos intervalos deben contener el mismo número de elementos. Sino, el resultado es indeterminado.*

```
std::vector<int> v1 = {1,2,3,4,5}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::reverse_copy( v1.begin(), v1.end(), v2.begin() );
//resultado: v2 = {5,4,3,2,1}
```

II-B-10 - rotate() y rotate_copy()

Complejidad: lineal.
rotate() efectúa una rotación sobre el orden de los elementos. *rotate()* toma los *n* primeros elementos y les pone al fin del intervalo.

```
std::vector<int> v = {1,2,3,4,5,6}; // no compila (cf. capítulo 2.0)
std::rotate( v.begin(), v.begin()+2, v.end() );
// resultado: v = {3,4,5,6,1,2}
```

rotate_copy() copia los elementos efectuando una rotación sober sus orden.

```
std::vector<int> v1 = {1,2,3,4,5,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::rotate( v1.begin(), v1.begin()+2, v1.end(), v2.begin() );
// resultado: v2 = {3,4,5,6,1,2}
```

II-B-11 - random_shuffle()

Complejidad: lineal.
Modifica, de modo aleatorio, la orden de los elementos de un intervalo.
Extraído de AlbumManager:

```
bool GetRandomAlbum( Album & album )
{
 // si la lista esta vacia, devolvemos false
 if ( main_list_.size() == 0 )
```

```
  return false;

 // Generación de la semilla
 srand( clock() );

 // modificación, de modo aleatorio, el orden de main_list_
 std::random_shuffle( main_list_.begin(), main_list_.end() );

 // copia del primero elemento de main_list_ en el resultado
 album = main_list_[0];
 return true;
}
```

II-B-12 - partition() y stable_partition()

Complejidad: lineal.
partition() modifica la orden de los elementos de manera que los elementos que corresponden a un criterio dado se encuentran delante.
Extraído de AlbumManager:

```
// HaveSameArtist es un predicado que devuelve true si el nombre del artista
// en parametro sea el mismo.
bool Album::HaveSameArtist(std::string artist) const { return artist_.GetName() == artist; }

// un nombre de artista
std::string artistName = "le nom d'un artiste";

// hacemos una partición de main_list_ en 2 partes: al principio, los discos de los que el artista corresponde
// en nombre del artista buscado, los otros después.
AlbumList::iterator cut =
 std::partition( main_list_.begin(),
main_list_.end(),
std::bind2nd( std::mem_fun_ref<bool, Album, std::string>(&Album::HaveSameArtist ), artistName) );
```

Complejidad: *stable_partition()* Es un algoritmo adaptativo: intenta asignar un buffer temporal, y su complejidad depende de la memoria disponible.
En lo peor de los casos (si ninguna memoria auxilliaire está disponible), hay en el peor de los casos N*log (N) cambios, donde N está el número de elementos en el contenedor. Y en el mejor caso (tan mucha memoria está disponible), complejidad es lineal. En todos los casos, el predicado es aplicado N la vez. *stable_partition()* hace la misma cosa que *partition()*, pero conserva la orden relativa de los elementos correspondientes o no al criterio.

```
// IsOdd es un predicado que deter,ina si un nombre es impa
bool IsOdd(int number){ return number&1;}

std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // no compila (cf. capítulo 2.0)
std::stable_partition( v1.begin(), v1.end(), IsOdd );
// resultado : v1 = {5,3,1,7,4,2,8,6}
// con el algoritmo partition(), habría dado : v1 = {5,3,1,7,2,8,4,6}
```

Vuelta al principio

II-C - Algoritmos de clasificación y operaciónes vinculadas

II-C-1 - algoritmos de clasificación

II-C-1-1 - sort()

Complejidad: N*log(N)
sort() clasifica los elementos de un intervalo.

```
std::vector<int> v = {3,2,4,1,5}; // no compila (cf. capítulo 2.0)
std::sort( v.begin(), v.end() );
// resultado: v = {1,2,3,4,5}
```

II-C-1-2 - stable_sort()

Complejidad: *stable_sort()* es un algoritmo adaptativo: intenta asignar un buffer temporal, y su complejidad depende de la memoria disponible.
En lo peor de los casos (si ninguna memoria auxilliaire está disponible), hay en el peor de los casos N*log (N)^2 comparaciones, donde N está el número de elementos en el contenedor. Y en el mejor caso (tan mucha memoria está disponible), complejidad es N*log(N).
Clasifica preservando la orden de los elementos iguales.

```
// Predicado que compara dos char sin tomar cuenta de la caja
bool CompareNoCase(char c1, char c2) { return std::toupper(c1) < std::toupper(c2); }

std::vector<char> s1 = { d,B,b,C,A,D,a,c } ; // no compila (cf. capítulo 2.0)
std::stable_sort(s1.begin(), s1.end(), CompareNoCase );
// resultado: s1 = {A,a,B,b,C,c,d,D}
```

II-C-1-3 - partial_sort()

Complejidad: aproximadamente N*log(N).
partial_sort() Clasifica un intervalo hasta que *n* elementos hubieran sido clasificados.

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // no compila (cf. capítulo 2.0)
std::partial_sort( v1.begin(), v1.begin() + 4, v1.end() );
// resultado: v1 = {1,2,3,4,5,8,7,6}
```

II-C-1-4 - partial_sort_copy()

Complejidad: aproximadamente N*log(N).
partial_sort_copy() copia los elementos an un orden clasificado.

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2(4);
std::partial_sort_copy( v1.begin(), v1.begin() + 4, v2.begin(), v2.end() );
// resultado: v2 = {1,3,4,5}
```

II-C-2 - nth_element()

Complejidad: lineal, por término medio.
Clasifica con arreglo a la enésima posición.
nth_element() es igual a *partial_sort()*, en el sentido donde esta función clasifica parcialmente un intervalo de elementos: arregla el intervalo [first, last) de manera que el elemento apuntado por el iterador *nth* sea el mismo que el elemento que debería estar en esta posición si el intervalo entero [first, last) había sido clasificado. Además, ningún elemento del intervalo [nth, last) es inferior a todos los elementos en el intervalo [first, last).

```
std::vector<int> v1 = {5,3,1,4,2,8,7,6}; // no compila (cf. capítulo 2.0)
std::nth_element( v.begin(), v.begin()+3, v.end() );
// resultado : v1 = {1,2,3,4,5,6,7,8}
```

En el ejemplo más arriba, el contenedor íntegramente esta clasificado. Esto respeta las especificaciones. Ciertamente por razones de eficacia, ciertas versiones del STL parecen implementar nth_element() como una clasificación simple en ciertas circunstancias.

II-C-3 - Búsqueda binaria

II-C-3-1 - lower_bound()

Complejidad: log(N).
lower_bound() encuentra el primer elemento superior o igual a un valor dado. Típicamente lo utilizamos para insertar un elemento en un intervalo clasificado para respetar la orden.

```
std::vector<int> v = {1,2,4,6,9}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator it = std::lower_bound( v.begin(), v.end(), 4 );
// resultado: it apunta en el tercero elemento de v (4)
```

II-C-3-2 - upper_bound()

upper_bound() encuentra el primer elemento superior a un valor dado.

```
std::vector<int> v = {1,2,4,6,9}; // no compila (cf. capítulo 2.0)
std::vector<int>::iterator it = std::upper_bound( v.begin(), v.end(), 4 );
// resultado: it apunta en le cuarto elemento de v (6)
```

II-C-3-3 - equal_range()

Complejidad: o(2 x ln N).
equal_range() intenta encontrar un elemento en un intervalo clasificado. Esta función devuelve un par de itérateurs i y j tales, que i es la primera posición donde el valor puede ser insertado sin modificar la orden del intervalo, y j es la última posición donde el valor puede ser insertado sin modificar la orden de él intervalo. Resulta de eso que cada elemento del intervalo [i, j) es equivalente al valor buscado.

```
std::vector<int> v = {1,2,4,4,6,7}; // no compila (cf. capítulo 2.0)
std::pair<std::vector<int>::iterator, std::vector<int>::iterator> newRange ;
newRange = std::equal_range(v.begin(), v.end(), 4) ;
// resultado : r.first apunta en al tercero elemento de v, f.second apunta en el quinto elemento de v (6)
```

II-C-3-4 - binary_search()

Complejidad: log(N).
binary_search() devuelve *true* si el intervalo contiene un elemento dado.

```
std::vector<int> v1 = {1,2,4,6}; // no compila (cf. capítulo 2.0)
bool b = std::binary_search( v1.begin(), v1.end(), 9 );
// resultado: b = false
```

II-C-4 - merge() y inplace_merge()

Complejidad: lineal.
merge() fusiona dos intervalos *I1* et *I2* en un tercero intervalo *I3*. *I1* y *I2* deben estar clasificados, y el resultado de la fusión estara igualmente clasificado.
Extraído de AlbumManager:

```
std::vector<int> v1 = {1,4,5,7}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,3,6,8}; // no compila (cf. capítulo 2.0)
std::vector v3; // Contenedor en el cual vamos a fusionar v1 y v2
std::merge( v1.begin(), v1.begin(), v2.begin(), v2.end(), back_inserter(v3) );
// resultado: v3 = {1,2,3,4,5,6,7,8}
```

Complejidad: *stable_sort()* es un algoritmo adaptativo: intenta asignar un buffer temporal, y su complejidad depende de la memoria disponible.
inplace_merge() combina 2 intervalos clasificados I1 y I2 del mismo contenedor C. Más precisamente, comienza por construir un intervalo I3 = [I1.begin, I2.end], luego efectúa una selección ascendiente sobre I3.
inplace_merge() es estable, lo que significa que la orden relativa de los intervalos iniciales es preservada.

```
std::vector<int> v = {1,3,5,2,4,5,6}; // no compila (cf. capítulo 2.0)
std::inplace_merge( v.begin(), v.begin()+3, v.end() );
// resultado: v = {1,2,3,4,5,5,6}
```

II-C-5 - Operaciones de "set" sobre intervalos clasificados

Complejidad: lineal.

II-C-5-1 - includes()

Estos algoritmos deben ser aplicados sobre intervalos clasificados.

includes() devuelve *true* si cada elemento de un intervalo también es un elemento de un otro intervalo.
Extraído de AlbumManager :

```
// Verificamos si una lista es incluida en la otra, a en cuyo caso,
// basta con devolver la que incluye el otro
if ( std::includes( main_list_.begin(), main_list_.end(), second_list_.begin(), second_list_.end() ) )
{
 return main_list_;
}
if ( std::includes( second_list_.begin(), second_list_.end(), main_list_.begin(), main_list_.end() ) )
{
 return second_list_;
}
```

II-C-5-2 - set_union()

set_union() efectúa la operación aritmética de unión sobre 2 intervalos clasificados.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // no compila (cf. capítulo 2.0)
std::vector<int> v3;
std::set_union( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultado: v3 = {1,2,2,3,3,4,4,6,6,8}
```

II-C-5-3 - set_intersection()

set_intersection() efectúa la operación aritmética de intersection sobre 2 intervalos clasificados.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // no compila (cf. capítulo 2.0)
std::vector<int> v3;
std::set_intersection( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultado: v3 = {2,2,6}
```

II-C-5-4 - set_difference()

set_difference() efectúa la operación aritmética de diferencia sobre 2 intervalos clasificados.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // no compila (cf. capítulo 2.0)
std::vector<int> v3;
std::set_difference( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultado: v3 = {1,3,3,4,4}
```

II-C-5-5 - set_symetric_difference()

set_symetric_difference() efectúa la operación aritmética de diferencia simétrica sobre 2 intervalos clasificados.

```
std::vector<int> v1 = {1,2,2,3,3,4,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,2,6,6,8}; // no compila (cf. capítulo 2.0)
std::vector<int> v3;
std::set_symetric_difference( v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v3) );
//resultado: v3 = {1,3,3,4,4,6,8}
```

II-C-6 - Montones (heap)

Un montón (heap) es una estructura organizada según los criterios siguientes:
No hay elemento estrictamente superior al primer elemento del montón.
El primer elemento del montón debe poder ser quitado (pop) en O(log (N))
Un elemento puede ser añadido (push) también en O (log (N))
Un ejemplo de utilización de los algoritmos "heap".

II-C-6-1 - make_heap()

Complejidad: lineal. En el peor de los casos, 3*N comparaisons.
make_heap() transforma un contenedor en montón.

```
std::vector<int> v = {3,4,5,1,4,3,8}; // no compila (cf. capítulo 2.0)
std::make_heap( v.begin(), v.end() );
//resultado: v = {8,4,6,4,3,1,3}
```

II-C-6-2 - push_heap()

Complejidad: Logarítmico.
push_heap() añade un elemento en un montón. De verdad, hace la misma cosa que *make_heap()* pero su complejidad es menor ya que el contenedor ya es un montón.

```
std::vector<int> v = {8,4,6,4,3,1,3}; // no compila (cf. capítulo 2.0)
v.push_back(9);
std::push_heap( v.begin(), v.end() );
//resultado: v = {9,8,6,4,3,1,3,4}
```

II-C-6-3 - pop_heap()

Complejidad: Logarítmico.
pop_heap() pop_heap toma el elemento más grande, entonces el primero, y lo pone en el fin del contenedor.

pop_heap() no suprime el elemento y entonces, no modifica el tamaño del contenedor

```
std::vector<int> v = {9,8,6,4,3,1,3,4}; // no compila (cf. capítulo 2.0)
std::pop_heap( v.begin(), v.end() );
//resultado: v = {8,4,6,4,3,1,3,9}
```

II-C-6-4 - sort_heap()

Complejidad: en el peor de los casos O(log(N)).
sort_heap() clasifica un montón.

```
std::vector<int> v = {9,8,6,4,3,1,3,4}; // no compila (cf. capítulo 2.0)
std::sort_heap( v.begin(), v.end() );
//resultado: v = {1,3,3,4,4,6,8,9}
```

II-C-7 - Mínimo y máximo

II-C-7-1 - min()

Devuelve el más pequeño de 2 elementos.

```
int num = std::min(3, 4);
//resultado: num = 3;
```

II-C-7-2 - max()

Devuelve el más grande de 2 elementos.

```
int num = std::max(3, 4);
//resultado: num = 4;
```

II-C-7-3 - min_element()

Complejidad: lineal
Devuelve el más pequeño elemento de un intervalo.

```
// IsNewerThan es un predicado que devuelve true si el año de salida del disco pasado en parametro
// es inferior al del álbum.
bool Album::IsNewerThan(const Album & other) const { return year_ < other.GetYear(); }

Album GetOldest()
{
 AlbumList::const_iterator oldest =
  std::min_element( main_list_.begin(), main_list_.end(), std::mem_fun_ref( &Album::IsNewerThan ) );

 return Album(*oldest);
}
```

II-C-7-4 - max_element()

Complejidad: lineal
Devuelve el elemento más grande de un intervalo.

```
// IsNewerThan es un predicado que devuelve true si el año de salida del disco pasado en parametro
// es inferior al del álbum.
bool Album::IsNewerThan(const Album & other) const { return year_ < other.GetYear(); }

Album GetMostRecent()
{
 AlbumList::const_iterator mostRecent =
  std::max_element( main_list_.begin(), main_list_.end(), std::mem_fun_ref( &Album::IsNewerThan ) );

 return Album(*mostRecent);
```

```
}
```

II-C-8 - lexicographical_compare()

Complejidad: lineal.
Devuelve si un intervalo es inferior a otro según una orden lexicográfica.

⚠ *Ambos intervalos comparados deben contener el mismo número de elementos. Si no es el caso, el resultado es indeterminado.*

```
std::vector<int> v1 = {1,3,4,5}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,3,4,5}; // no compila (cf. capítulo 2.0)
std::lexicographical_compare( v1.begin(), v1.end(), v2.begin(), v2.end() ); // devuelve true
```

II-C-9 - algoritmos de permutación

II-C-9-1 - next_permutation()

Complejidad: lineal.
next_permutation() transforma un intervalo para que se haga la próxima permutación más grande y lexicográfica. Devuelve *true* si una permutación se efectuó, *false* si no.

```
std::vector<int> v1 = {5,3,1,4,2}; // no compila (cf. capítulo 2.0)
std::next_permutation( v1.begin(), v1.end() );
// resultado : v1 = {5,3,2,1,4}
```

II-C-9-2 - prev_permutation()

prev_permutation() transforma un intervalo para que se haga la próxima permutación más pequeña y lexicográfica. Devuelve *true* si una permutación se efectuó, *false* si no.

```
std::vector<int> v1 = {5,3,1,4,2}; // no compila (cf. capítulo 2.0)
std::prev_permutation( v1.begin(), v1.end() );
// resultado : v1 = {5,3,1,2,4}
```

Vuelta al principio

II-D - Algorithmos numéricos

Los algoritmos numéricos estan definidos en la cabecera *<numeric>*.

II-D-1 - accumulate()

Complejidad: lineal.
Combina todos los valores de los elementos (efectúa la suma, producto, etc.)
La función *accumulate()* utiliza un valor inicial. Hay varias razones para las cuales este valor es importante, la primera es que esto permite obtener siempre un resultado válido (particularmente en caso de que el intervalo está vacío).
Usted encontrará un ejemplo concreto de utilización de *accumulate()* en el código fuente de AlbumManager, pero es utilizado en un contexto un poco particular, entonces prefiero darle un ejemplo simple aquí:

```
// utilización de accumulate() sin función binaria
std::vector<int> v = {1,2,3,4,5}; // no compila (cf. capítulo 2.0)
int r = std::accumulate( v.begin(), v.end(), 0 ); // Aquí 0 corresponde al valor inicial
```

```
// resultado: r = 0 + 1 + 2 + 3 + 4 + 5 = 15

// Mult es una función binaria que multiplica dos enteros y devuelve el resultado de esta multiplicación:
int Mult(const int a, const int b){ return a*b; }

// utilización de accumulate() con una función binaria:
r = std::accumulate( v.begin(), v.end(), 1, Mult ); // Aquí 1 corresponde al valor inicial
// Si este valor hubiera sido 0, el resultado habría sido 0.
// resultado: r = 1 * 1 * 2 * 3 * 4 * 5 = 120
```

II-D-2 - inner_product()

Complejidad: lineal.
Combina todos los elementos de dos intervalos.

```
std::vector<int> v1 = {1,2,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2 = {2,1,2,-1}; // no compila (cf. capítulo 2.0)
int r = std::inner_product( v1.begin(), v1.end(), v2.begin(), 0
 ); // Aquí 1 corresponde al valor inicial
// resultado : r = 0 + 1*2 + 2*1 + 4*2 + 6*-1 =
```

II-D-3 - adjacent_difference()

Complejidad: lineal.
Combina cada elemento con su predecesor.

```
std::vector<int> v1 = {1,2,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() );
std::adjacent_difference( v1.begin(), v1.end(), v2.begin() );
// resultado : v2 = {1,1,2,2}
```

II-D-4 - partial_sum()

Complejidad: lineal.
Combina cada elemento con cada uno de sus predecesores.

```
std::vector<int> v1 = {1,2,4,6}; // no compila (cf. capítulo 2.0)
std::vector<int> v2( v1.size() ) ;
std::adjacent_difference( v1.begin(), v1.end(), v2.begin() );
// resultado : v2 = {1,3,7,13}
```

Un ejemplo de utilización intensiva de los algoritmos: AlbumManager

Este programa administra una lista de álbumes. Va a leer 2 archivos de texto y a crear 2 listas de álbumes: una lista principal (*ain_list_*) y una lista secundaria (*second_list_*).
Luego, vamos a poder aplicar toda una variedad de algoritmos sobre estas dos listas.
El fin de este pequeño programa únicamente es implementar un máximo de algoritmos diferentes. Entonces, desde luego, la utilización de estos algoritmos no es siempre adaptada perfectamente, y a menudo, las mejores soluciones habrían sido posibles.
Ademas, no intenté optimizar este programa.


Los comentarios de este program son en frances.

Annexos

V-A - Fonctionoides

No utilizo fonctionoides en el código fuente de ejemplo proporcionado con este artículo, y no es impescindible saber lo que es para utilizar los algoritmos de la STL. Sin embargo, esta técnica permite resolver hábilmente problemas que se ponen a veces cuando se utiliza estos algoritmos. Entonces es interesante hablar un poco de esos. Se trata de un concepto inherente a la poo (programación orientada objeto) que permite utilizar un mecanismo que reúne las ventajas de los functores y de la herencia del C++.
Tomemos un ejemplo para ayudarnos a entender el funcionamiento de este animal extraño.
Tomemos una clase *Coche*, proveída de una función *GetNbRuedas()* que devuelve el número de ruedas del vehículo. En nuestro programa, tenemos un arrayde *Coche* que representa nuestro parque de vehículo, y en un momento dado, necesitamos saber el número total de ruedas de nuestro array de coches.

```
// estructura Coche
struct Coche
{
 int GetNbRuedas() const {return 4 ;}
};

// un array de coches
std::vector<Coche> v;

// Una implementación posible para recuperar el número total de ruedas:
int ComputeWheelNumber()
{
 // Creamos un vector que contiene el número de rueda de cada coche
 std::vector<int> nbRuedas( v.size() );
 std::transform( v.begin(), v.end(), nbRuedas.begin(), std::mem_fun_ref( &Coche::GetNbRuedas ) );

 // Devolvemos la suma de todos los elementos de este vector
 return std::accumulate( nbRuedas.begin(), nbRuedas.end(), 0 );
}
```

Ver descripción de las funciones **transform()** et **accumulate()**

Luego, avanzamos en nuestro programa, y llega un momento cuando se debe añadir motocicletas en nuestro parque de vehículos. Un modo de resolver el problema de la cuenta de ruedas es utilizar fonctionoides. Esto consiste en crear una clase madre (a la que se llamará *Vehiculo*) que contiene una función virtual pura (*GetNbRuedas()*) y derivar de eso 2 clases chicas (*Coche* y *Motocicleta*) que implementarán esta función según sus especificidades:

```
clase Vehiculo
{
public:
 virtual int GetNbRuedas() const = 0 ;
};

class Coche : public Vehiculo
{
public:
 int GetNbRuedas() const {return 4 ;}
};

Class Motocicleta : public Vehiculo
{
public:
 int GetNbRuedas() const {return 2 ;}
};
```

Nuestro vector *v* será ahora un array de punteros de Vehiculo. En efecto, dado que *Vehiculo* es una clase abstracta, no puede ser instanciada, no podemos declarar un vector simple de *Vehiculo*:

```
std::vector<Vehiculo*> v;
```

Uno de los aspectos interesantes de esta técnica es que casi no necesitamos modificar la función *ComputeWheelNumber()*. En efecto, la única modificación que tenemos que efectuar está sobre el adaptador de función *mem_fun()* (ver capítulo I-C **chapitre suivant**), y viene por el hecho de que *v* es ahora un vector de apuntadores:

```
int ComputeWheelNumber()
{
 // Creamos un vector que contiene el número de rueda de cada coche
 std::vector<int> nbRuedas( v.size() );
 std::transform( v.begin(), v.end(), nbRuedas.begin(), std::mem_fun( & Vehicule::GetNbRuedas ) );

 // Devolvemos la suma de todos los elementos de este vector
 return std::accumulate( nbRuedas.begin(), nbRuedas.end(), 0 );
}
```

Ver descripción de las funciones **transform()** et **accumulate()**

En realidad, lo que es verdaderamente interesante en el sistema del fonctionoides es poder aliar la potencia de los functores que permite pasarles parámetros a los constructores (con el fin de servirse de eso en las funciones miembros) y la fuerza de la herencia y de la sobrecarga. Sin embargo, no es el sujeto de este artículo, también no hablaré más de esto.

Referencias:
C++ FAQ Lite

V-B - Referencias

Página de cppreference sobre la lib algorithm
Sitio SGI sobre la STL
C++ FAQ Lite
Curso de C++ de ZATOR Systems

Página inicial

Vuelta al principio