Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 4.2	Unidad 4.2
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

STL (Standard Template Library)

MARCO TEÓRICO

TEMPLATES – FUNCIONES LIBRES – STL – PROGRAMACIÓN GENÉRICA

(Se recomienda su lectura para poder hacer los ejercicios)

La biblioteca de plantillas estándar (STL) de C++

La biblioteca de plantillas estándar (STL) es un conjunto de clases de plantilla de C++ para proporcionar estructuras y funciones de datos de programación comunes, como listas, pilas, matrices, etc. Es una biblioteca de clases de contenedores, algoritmos e iteradores. Es una biblioteca generalizada y por lo tanto, sus componentes están parametrizados. Un conocimiento práctico de las clases de plantilla es un requisito previo para trabajar con STL.

STL tiene cuatro componentes

- Algoritmos
- Contenedores
- Funciones
- Iteradores

Algoritmos

El algoritmo de encabezado define una colección de funciones especialmente diseñadas para ser utilizadas en rangos de elementos. Actúan sobre los contenedores y proporcionan medios para diversas operaciones para el contenido de los contenedores.

Contenedores

Los contenedores o clases de contenedor almacenan objetos y datos. Hay en total siete clases de contenedores estándar de "primera clase" y tres clases de adaptadores de contenedores y solo siete archivos de encabezado que proporcionan acceso a estos contenedores o adaptadores de contenedores.

- Contenedores de secuencia: implementar estructuras de datos a las que se pueda acceder de forma secuencial.
  - vector
  - lista
  - deque
  - Matrices
  - forward_list(Introducido en C++11)
- Adaptadores de contenedores: proporcionan una interfaz diferente para contenedores secuenciales.
  - cola

- priority_queue
- pila

● Contenedores asociativos: implementar estructuras de datos ordenadas que se puedan buscar rápidamente (complejidad O(log n)).

- poner
- multiconjunto
- mapa
- multimapa

● Contenedores asociativos no ordenados: implemente estructuras de datos desordenadas que se puedan buscar rápidamente

- unordered_set (introducido en C++11)
- unordered_multiset (introducido en C++11)
- unordered_map (Introduced in C++11)
- unordered_multimap (Introduced in C++11)

Funciones

El STL incluye clases que sobrecargan el operador de llamada a la función. Las instancias de tales clases se denominan objetos de función o funtores. Los funtores permiten personalizar el funcionamiento de la función asociada con la ayuda de parámetros a pasar.

● Funtors

Iteradores

Como su nombre indica, los iteradores se utilizan para trabajar en una secuencia de valores. Son la característica principal que permite la generalidad en STL.

● Iteradores

Biblioteca de utilidades

Definido en el encabezado <utilidad>.

● pair

Vector en C++ STL

Los vectores son los mismos que las matrices dinámicas con la capacidad de cambiar el tamaño automáticamente cuando se inserta o elimina un elemento, y su almacenamiento es manejado automáticamente por el contenedor. Los elementos vectoriales se colocan en un almacenamiento contiguo para que se pueda acceder a ellos y recorrerlos mediante iteradores. En los vectores, los datos se insertan al final. La inserción al final lleva un tiempo diferencial, ya

que a veces es posible que sea necesario extender la matriz. La eliminación del último elemento solo lleva tiempo constante porque no se produce ningún cambio de tamaño. Insertar y borrar al principio o en el medio es lineal en el tiempo.

Ciertas funciones asociadas con el vector son: Iteradores

1. begin() – Devuelve un iterador que apunta al primer elemento del vector.
2. end() – Devuelve un iterador que apunta al elemento teórico que sigue al último elemento del vector.
3. rbegin(): devuelve un iterador inverso que apunta al último elemento del vector (inicio inverso). Se mueve del último al primer elemento
4. rend() – Devuelve un iterador inverso que apunta al elemento teórico que precede al primer elemento del vector (considerado como extremo inverso)
5. cbegin(): devuelve un iterador constante que apunta al primer elemento del vector.
6. cend() – Devuelve un iterador constante que apunta al elemento teórico que sigue al último elemento del vector.
7. crbegin(): devuelve un iterador inverso constante que apunta al último elemento del vector (inicio inverso). Se mueve del último al primer elemento
8. crend() – Devuelve un iterador inverso constante que apunta al elemento teórico que precede al primer elemento del vector (considerado como extremo inverso)

Capacidad

1. size(): devuelve el número de elementos del vector.

2. max_size(): devuelve el número máximo de elementos que puede contener el vector.

3. capacity(): devuelve el tamaño del espacio de almacenamiento asignado actualmente al vector expresado como número de elementos.

4. resize(n): cambia el tamaño del contenedor para que contenga 'n' elementos.

5. empty(): devuelve si el contenedor está vacío.

6. shrink_to_fit() – Reduce la capacidad del contenedor para adaptarse a su tamaño y destruye todos los elementos más allá de la capacidad.

7. reserve() – Solicita que la capacidad vectorial sea al menos suficiente para contener n elementos.

Iteradores vectoriales de C++

Los iteradores vectoriales se utilizan para apuntar a la dirección de memoria de un elemento vectorial. De alguna manera, actúan como punteros en C++.

Podemos crear iteradores vectoriales con la sintaxis

```
vector<T>::iterator iteratorName;
```

Por ejemplo, si tenemos 2 vectores de y tipos, entonces necesitaremos 2 iteradores diferentes correspondientes a sus tipos:intdouble

```
// iterator for int vector
vector<int>::iterator iter1;

// iterator for double vector
vector<double>::iterator iter2;
```

Inicializar iteradores vectoriales

Podemos inicializar iteradores vectoriales usando las funciones y.begin()end()
1. función begin()
La función devuelve un iterador que apunta al primer elemento del vector. Por ejemplobegin()

```
vector<int> num = {1, 2, 3};
vector<int>::iterator iter;

// iter points to num[0]
iter = num.begin();
```

2. función end()
La función apunta al **elemento teórico** que viene **después** del elemento final del vector. Por ejemploend()

```
// iter points to the last element of num
iter = num.end() - 1;
```

Aquí, debido a la naturaleza de la función, hemos utilizado el código para apuntar al último elemento del vector, es decir, .end()num.end() - 1numnum[2]

Ejemplo: Iteradores vectoriales de C++

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
 vector<int> num {1, 2, 3, 4, 5};

 // declare iterator
 vector<int>::iterator iter;
```

```
 // initialize the iterator with the first element
 iter = num.begin();

 // print the vector element
 cout << "num[0] = " << *iter << endl;

 // iterator points to the 3rd element
 iter = num.begin() + 2;
 cout << "num[2] = " << *iter;

 // iterator points to the last element
 iter = num.end() - 1;
 cout << "num[4] = " << *iter;

 return 0;
}
Ejecutar código
```

Salida

```
num[0] = 1
num[2] = 3
num[4] = 5
```

En este programa, hemos declarado un iterador vectorial para usarlo con el vector .intiternum

```
// declare iterator
vector<int>::iterator iter;
```

Luego, inicializamos el iterador al primer elemento del vector usando la función.begin()

```
// initialize the iterator with the first element
iter = num.begin();
```

Luego, imprimimos el elemento vectorial **desrreferenciando** el iterador:

```
// print the vector element
cout << "num[0] = " << *iter << endl;
```

Luego, imprimimos el 3er elemento del vector cambiando el valor de a .iternum.begin() + 2
Finalmente, imprimimos el último elemento del vector usando la función.end()

Ejemplo: Iterar a través de vector usando iteradores

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main() {
 vector<int> num {1, 2, 3, 4, 5};

 // declare iterator
 vector<int>::iterator iter;

 // use iterator with for loop
 for (iter = num.begin(); iter != num.end(); ++iter) {
  cout << *iter << "  ";
 }

 return 0;
}
Ejecutar código
```

Salida

```
1 2 3 4 5
```

Aquí, hemos utilizado un bucle para inicializar e iterar el iterador desde el principio del vector hasta el final del vector utilizando las funciones y.foriterbegin()end()

```
// use iterator with for loop
for (iter = num.begin(); iter != num.end(); ++iter) {
  cout << *iter << "  ";
}
```

Map en la biblioteca de plantillas estándar (STL) de C++

Los mapas son contenedores asociativos que almacenan elementos de manera mapeada. Cada elemento tiene un valor clave y un valor asignado. No hay dos valores asignados que puedan tener los mismos valores clave.

Lista de todas las funciones del mapa

Función	Definición
mapa::insertar()	Inserte elementos con una clave concreta en el contenedor de mapas.
map:: count()	Devuelve el número de coincidencias al elemento con el valor clave 'g' en el mapa.

Función	Definición
mapa equal_range()	Devuelve un iterador de pares. El par se refiere a los límites de un rango que incluye todos los elementos en el contenedor que tienen una clave equivalente a k.
borrado de mapa()	Se utiliza para borrar elementos del contenedor.
mapa rend()	Devuelve un iterador inverso que apunta al elemento teórico justo antes del primer par clave-valor del mapa (que se considera su extremo inverso).
mapa rbegin()	Devuelve un iterador inverso que apunta al último elemento del mapa.
mapa find()	Devuelve un iterador al elemento con clave-valor 'g' en el mapa si se encuentra, de lo contrario devuelve el iterador al final.
mapa crbegin() y crend()	crbegin() devuelve un iterador inverso constante que hace referencia al último elemento del contenedor de mapas. crend() devuelve un iterador inverso constante que apunta al elemento teórico antes del primer elemento del mapa.
mapa cbegin() y cend()	cbegin() devuelve un iterador constante que hace referencia al primer elemento del contenedor de mapas. cend() devuelve un iterador constante que apunta al elemento teórico que sigue al último elemento del multimapa.
mapa emplace()	Inserta la clave y su elemento en el contenedor de mapas.

Función	Definición
mapa max_size()	Devuelve el número máximo de elementos que puede contener un contenedor de mapas.
mapa upper_bound()	Devuelve un iterador al primer elemento que es equivalente al valor asignado con clave-valor 'g' o definitivamente irá tras el elemento con clave-valor 'g' en el mapa
operador de mapa=	Asigna el contenido de un contenedor a un contenedor diferente, reemplazando su contenido actual.
mapa lower_bound()	Devuelve un iterador al primer elemento que es equivalente al valor asignado con clave-valor 'g' o definitivamente no irá antes que el elemento con clave-valor 'g' en el mapa.
map emplace_hint()	Inserta la clave y su elemento en el contenedor de mapas con una sugerencia determinada.
map value_comp()	Returns the object that determines how the elements in the map are ordered (‘<‘ by default).
map key_comp()	Returns the object that determines how the elements in the map are ordered (‘<‘ by default).
map::size()	Returns the number of elements in the map.
map::empty()	Returns whether the map is empty

Función	Definición
map::begin() and end()	begin() returns an iterator to the first element in the map. end() returns an iterator to the theoretical element that follows the last element in the map
map::operator[]	Este operador se utiliza para hacer referencia al elemento presente en la posición dada dentro del operador.
map::clear()	Quita todos los elementos del mapa.
map::at() y map::swap()	La función at() se utiliza para devolver la referencia al elemento asociado a la clave k. la función swap() se utiliza para intercambiar el contenido de dos mapas, pero los mapas deben ser del mismo tipo, aunque los tamaños pueden diferir.

unordered_map en C++ STL

unordered_map es un contenedor asociado que almacena elementos formados por la combinación de clave-valor y un valor asignado. El valor de clave se utiliza para identificar de forma única el elemento y el valor asignado es el contenido asociado a la clave. Tanto la clave como el valor pueden ser de cualquier tipo predefinidos o definidos por el usuario.

Internamente unordered_map se implementa utilizando Hash Table, la clave proporcionada para mapear se hastizan en índices de una tabla hash, por lo que el rendimiento de la estructura de datos depende mucho de la función hash, pero en promedio, el costo de búsqueda, inserción y eliminación de la tabla hash es O(1).

Nota: En el peor de los casos, su complejidad temporal puede ir de O(1) a O(n2), especialmente para números primos grandes. Puede leer más sobre esto en cómo usar unordered_map eficientemente en c. En esta situación, es muy recomendable utilizar un mapa en su lugar para evitar // C++ program to demonstrate functionality of unordered_map

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
  // Declaring umap to be of <string, int> type
  // key will be of string type and mapped value will
```

```
  // be of int type
  unordered_map<string, int> umap;

  // inserting values by using [] operator
  umap["Green"] = 10;
  umap["Practice"] = 20;
  umap["Contribute"] = 30;

  // Traversing an unordered map
  for (auto x : umap)
  cout << x.first << " " << x.second << endl;

}
```

```
Salida:
Contribute 30

Green 10

Practice 20
```

unordered_map vs unordered_set:

En unordered_set, solo tenemos clave, sin valor, estas se utilizan principalmente para ver presencia/ausencia en un conjunto. Por ejemplo, considere el problema de contar las frecuencias de palabras individuales. No podemos usar unordered_set (o conjunto) ya que no podemos almacenar recuentos.

unordered_map vs mapa:

mapa (como conjunto) es una secuencia ordenada de claves únicas, mientras que en unordered_map clave se puede almacenar en cualquier orden, por lo que no está ordenada.

El mapa se implementa como una estructura de árbol equilibrada, por lo que es posible mantener el orden entre los elementos (por recorrido de árbol específico). La complejidad temporal de las operaciones de mapa es O(log n), mientras que para unordered_map, es O(1) en promedio.

Métodos en unordered_map

Hay muchas funciones disponibles que funcionan en unordered_map. los más útiles de ellos son: operador =, operador [], vacío y tamaño para la capacidad, comienzo y fin para el iterador, buscar y contar para la búsqueda, insertar y borrar para la modificación.

```
//Ejemplo de Código
// C++ program to demonstrate functionality of unordered_map
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
```

```
    // Declaring umap to be of <string, double> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, double> umap;

    // inserting values by using [] operator
    umap["PI"] = 3.14;
    umap["root2"] = 1.414;
    umap["root3"] = 1.732;
    umap["log10"] = 2.302;
    umap["loge"] = 1.0;

    // inserting value by insert function
    umap.insert(make_pair("e", 2.718));

    string key = "PI";

    // If key not found in map iterator to end is returned
    if (umap.find(key) == umap.end())
        cout << key << " not found\n\n";

    // If key found then iterator to that key is returned
    else
        cout << "Found " << key << "\n\n";

    key = "lambda";
    if (umap.find(key) == umap.end())
        cout << key << " not found\n";
    else
        cout << "Found " << key << endl;

    // iterating over all value of umap
    unordered_map<string, double>:: iterator itr;
    cout << "\nAll Elements : \n";
    for (itr = umap.begin(); itr != umap.end(); itr++)
    {
        // itr works as a pointer to pair<string, double>
        // type itr->first stores the key part and
        // itr->second stores the value part
        cout << itr->first << " " << itr->second << endl;
    }
}
```

Métodos de unordered_map :

- at(): Esta función en C++ unordered_map devuelve la referencia al valor con el elemento como clave k.
- begin(): devuelve un iterador que apunta al primer elemento del contenedor del contenedor unordered_map contenedor
- end(): Devuelve un iterador que apunta a la posición más allá del último elemento del contenedor en el contenedor de unordered_map.
- bucket(): Devuelve el número de bucket donde se encuentra el elemento con la clave k en el mapa.
- bucket_count: bucket_count se utiliza para contar el no total. de cubos en el unordered_map. No se requiere ningún parámetro para pasar a esta función.
- bucket_size: Devuelve el número de elementos de cada bucket del unordered_map.
- count(): Cuenta el número de elementos presentes en un unordered_map con una clave determinada.
- equal_range: Devuelve los límites de un rango que incluye todos los elementos del contenedor con una clave que se compara igual a k.
- find(): Devuelve el iterador al elemento.
- empty(): comprueba si el contenedor está vacío en el contenedor unordered_map.
- erase(): borrar elementos del contenedor en el contenedor unordered_map.

obtener un error TLE.

Marco Práctico

1. Ejercicio STL vector de C++
   a. Escribe un programa que defina un vector de enteros.
   b. Inserte dos elementos en un vector.
   c. Imprima el contenido vectorial utilizando el bucle basado en rangos.
   d. Borre el segundo elemento del vector.
   e. Imprima el contenido vectorial utilizando el bucle basado en rangos.
   f. Borre el rango de 3 elementos a partir del principio del vector.
   g. Imprima el contenido vectorial utilizando el bucle basado en rangos.
   h. Escriba un programa que busque un elemento vectorial utilizando la función de algoritmo std::find().
   i. Si se ha encontrado el elemento, elimínelo.
   j. Imprima el contenido vectorial.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v;                      // 1. Vector vacío

    v.push_back(10); v.push_back(20);   // 2. Insertar 2 elementos
    v.push_back(30); v.push_back(40);
    v.push_back(50);

    // 3. Imprimir con range-for
    cout << "Vector: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    v.erase(v.begin() + 1);             // 4. Borrar 2do elemento (índice 1)

    // 5. Imprimir
    cout << "Sin 2do: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    v.erase(v.begin(), v.begin() + 3); // 6. Borrar primeros 3

    // 7. Imprimir
    cout << "Sin primeros 3: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    auto it = find(v.begin(), v.end(), 50); // 8. Buscar 50
    if (it != v.end()) {
        v.erase(it);                        // 9. Eliminar si existe
        cout << "50 eliminado\n";
    }

    // 10. Imprimir final
    cout << "Final: ";
    for (int x : v) cout << x << " ";
    cout << endl;
}
```

2. STL Vector & Iterator

a. Utilizar el Ejemplo anterior e Implementar un Vector con un Iterador para mostrar su contenido.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<string> productos = {"Laptop", "Mouse", "Teclado", "Monitor"};

    cout << "Productos: ";
    for (auto it = productos.begin(); it != productos.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
}
```

3. Implementar un Map en C++

a. contenedor de mapa vacío

b. insertar elementos en orden aleatorio

c. imprimiendo mapa Map1

d. asignando los elementos de Map1 a Map2

e. imprimir todos los elementos del mapa Map2

f. eliminar todos los elementos hasta

g. elemento con clave=3 en Map2

h. eliminar todos los elementos con clave = 4

i. límite inferior y límite superior para el mapa Map1 clave = 5

```cpp
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<int, string> m1;                // 1. Map vacío

    m1[5] = "Laptop";                   // 2. Insertar en orden aleatorio
    m1[1] = "Mouse";
    m1[3] = "Teclado";
    m1[4] = "Monitor";

    cout << "Map1:\n";                  // 3. Imprimir m1
    for (auto& p : m1) cout << p.first << ": " << p.second << endl;

    map<int, string> m2 = m1;           // 4. Asignar m1 → m2

    cout << "\nMap2:\n";                // 5. Imprimir m2
    for (auto& p : m2) cout << p.first << ": " << p.second << endl;

    m2.erase(m2.begin(), m2.find(3));   // 6-7. Eliminar hasta clave=3 (excluye 3)
    m2.erase(4);                        // 8. Eliminar clave=4

    auto low = m1.lower_bound(5);       // 9. Límite inferior clave=5
    auto up = m1.upper_bound(5);        //    Límite superior clave=5
    cout << "\nClave 5: " << low->second << endl;
}
```

4. Implementar un unordermap que cuente la cantidad de repetición de palabrasde la frase:

"Existen dos tipos de lenguajes de programación: por un lado, aquellos de los que la gente se queja todo el rato; por otro, los que nadie utiliza." — Bjarne Stroustrup

```cpp
#include <iostream>
#include <unordered_map>
#include <string>
#include <sstream>
using namespace std;

int main() {
    string frase = "Existen dos tipos de lenguajes de programacion: por un lado, "
                   "aquellos de los que la gente se queja todo el rato; por otro, "
                   "los que nadie utiliza.";

    unordered_map<string, int> contador;  // unordered_map
    stringstream ss(frase);
    string palabra;

    while (ss >> palabra) {
        // Limpiar puntuación básica
        if (!palabra.empty() && ispunct(palabra.back()))
            palabra.pop_back();
        contador[palabra]++;
    }

    cout << "Frecuencia de palabras:\n";
    for (auto& p : contador) {
        cout << p.first << ": " << p.second << endl;
    }
}
```

Nota: Por disposición Ministerial (Resolución de Carrera), los temas: Pilas, Listas, Colas, Arboles y Grafos corresponden al Algoritmos y Estructuras de Datos III (ver resolución IF-2019-36021554-GDEBA-CPEYTDGCYE), por tal motivo trataremos estos temas (y otros) el año próximo.

Lic. Oemig José Luis.