Contenedores de C++

Sitio: Centros - Cádiz
Curso: Programación
Libro: Contenedores de C++
Imprimido por: Barroso López, Carlos
Día: martes, 21 de mayo de 2024, 23:28

Tabla de contenidos

1. Definición y tipos

1. Definición

Un contenedor en C++ es un estructura que almacena una **colección de objetos**, normalmente del mismo tipo.

Los contenedores son **plantillas** de clase, por tanto, cuando se declara una variable del contenedor, se debe especificar el tipo de los elementos que se incluirán en el contenedor.

```
contenedor<tipo_dato> identificador
```

Características

- Gestiona sus necesidades de **almacenamiento de forma dinámica** y transparente al programador.
- Permite el ***acceso, inserción y borrado*** de elementos de **forma controlada**.
- **Conoce** el número de elementos incluidos en el contenedor en cada momento (**tamaño**).

Funciones miembro

Los contenedores proporcionan una serie de funciones miembro que permiten operar con ellos: *insertar y borrar elementos, acceder a su contenido, conocer su tamaño, etc.*

```
identificador.funcion_miembro(parametros)
```

2. Clasificación y tipos

La ***biblioteca estándar de C++***, también conocida como STL (*Standard Template Library*, biblioteca de plantillas estándar), proporciona una colección de:

- **Contenedores** de diferentes tipos.
- **Funciones** para su manejo (miembro y no miembro).
- **Algoritmos** para múltiples propósitos: *buscar, contar, comparar, copiar, intercambiar, seleccionar u ordenar elementos.*

El uso de esta biblioteca permite a los programadores implementar fácilmente las **estructuras de datos más habituales en programación**.

Los contenedores de la biblioteca estándar se pueden clasificar en tres categorías:

2.1. Contenedores de secuencia

Los elementos mantienen el orden en el que se han insertado.

Contenedor 'vector' (Array dinámico)

```
vector<T>
```

Se comporta como un array pero puede crecer automáticamente según sea necesario. Su acceso es aleatorio y se almacena de forma contigua.

Contenedor 'deque' (Cola de dos extremos)

```
deque<T>
```

Permite **inserciones y eliminaciones rápidas** al principio y al final del contenedor. Comparte las ventajas de acceso aleatorio y longitud flexible de vector, pero no es contiguo.

Contenedor 'list' (Lista enlazada)

```
list<T>
```

Lista doblemente enlazada que permite el acceso bidireccional e inserciones y eliminaciones rápidas en cualquier parte del contenedor, pero no permite tener acceso de forma aleatoria a un elemento.

```
forward_list<T>
```

Versión de lista simplemente enlazada.

2.2. Contenedores asociativos

A diferencia de los secuenciales, el orden en el que se insertan los elementos es indiferente. Los elementos mantienen un orden predefinido, o bien, no mantienen ningún orden.

Contenedor map (Diccionario)

```
map<T>
```

Cada elemento consta de un par de **clave/valor**, manteniendo sus elementos ordenados por la clave. No permite claves repetidas.

```
unordered_map<T>
```

Versión del contenedor map no ordenada.

Contenedor set

```
set<T>
```

Conjunto de elementos únicos y ordenados (el propio valor es la clave).

```
unordered_set<T>
```

Versión del contenedor set no ordenada.

Tanto *map* como *set* permiten solo una instancia de una clave o elemento en el contenedor. Si se requieren varias instancias de elementos, se debe utilizar **multimap** o **multiset**.

Además existen sus versiones no ordenadas: unordered_multimap y unordered_multiset

2.3. Adaptadores de contenedor

Un adaptador de contenedor es una variación de un contenedor completo como los anteriores, que restringe la interfaz para una mayor simplicidad y claridad, proporcionando una **interfaz específica** para una estructura de datos determinada.

Contenedor queue (Cola)

```
queue<T>
```

Sigue la semántica FIFO (primero en entrar, primero en salir).

```
priority_queue<T>
```

Los elementos de la cola se ordenan según su **prioridad**.

Contenedor stack (Pila)

```
stack<T>
```

Sigue la semántica LIFO (último en entrar, primero en salir).

3. Elección del contenedor adecuado

Muchos de los contenedores de la biblioteca estándar comparten ciertas características o funcionalidades.

Por tanto, la decisión de qué tipo de contenedor utilizar para una necesidad específica, no solo dependerá de sus características y las funcionalidades que ofrece, sino también de la **eficiencia** (complejidad) de algunas **de sus operaciones**, como por ejemplo:

- *Acceso a los elementos*
- *Inserción y eliminación de elementos*
- *Búsqueda de un elemento*
- *Recorrido de la estructura*
- *Etc.*

Para ello, es importante conocer qué tipo de **operaciones se realizarán con mayor frecuencia**, y de este modo, elegir el contenedor que mejor rendimiento ofrezca.

2. Contenedor Vector

1. Inconvenientes de los arrays estilo C

Un array convencional tiene algunos inconvenientes:

- No contiene información sobre su tamaño.
- No controla el acceso fuera de sus límites.
- Es de tamaño fijo, por tanto, es necesario reservar memoria suficiente de antemano, con el consiguiente desperdicio que esto conlleva.

2. Definición

Un contenedor `vector` es una implementación de un array dinámico que permite almacenar una colección de objetos del mismo tipo.

Al igual que con los vectores estilo C:

- Se puede acceder a los elementos del contenedor vector usando un índice y el **operador** de indexación `[]`.
- Los elementos ocupan posiciones contiguas en memoria.

Este contenedor está definido en la **biblioteca <vector>**.

Sintaxis:

```
vector<T> identificador;
```

donde `T`, es la plantilla que indica el tipo de dato.

Ejemplos:

```
vector<int> v;           // Vector de enteros
vector<string> cadenas;  // Vector de cadenas
```

3. Inicialización

```
vector<T> v{x, y, z};  // (Asignación) Equivale a vector<T> v = {x, y, z};
```

```
vector<T> v(n, valor)  // (Relleno) 'n' elementos inicializados a 'valor' (o valor nulo si se omite)
```

```
vector<T> v(w);  // (Copia) Copia los elementos del vector w
```

4. Funciones miembro o métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Tamaño		
size()	Devuelve el número de elementos del vector.	R: size_t **
empty()	Chequea si el contenedor está vacío.	R: true/false
Acceso a elementos	(Devuelven una referencia al elemento)	
Operador [índice]	Acceso a un elemento mediante su índice.	P1: índice (size_t)

at(índice)	Acceso a un elemento mediante su índice, de forma controlada.	P1: índice (size_t)
front()	Acceso al primer elemento.	
back()	Acceso al último elemento.	

Modificadores

push_back(elemento)	Inserta un elemento al final del vector.	p1: elemento de tipo T a insertar.
pop_back()	Elimina el último elemento del vector (no verifica si está vacío).	
assign(elementos)	Asigna nuevos elementos al vector.	Constructores: Asignación y Relleno
clear()	Borra todo el contenido.	

** `size_t` es un tipo de dato **entero sin signo** definido en la biblioteca estándar y utilizado para representar tamaños y recuentos. Su tamaño puede variar en función del sistema donde se está ejecutando el programa.

3. Uso en funciones

1. Contenedor como parámetro de una función

Al igual que cualquier otro tipo de parámetro, un contenedor se puede pasar como argumento de una función **por valor o por referencia**:

- **Por valor**: se copian todos los elementos del contenedor en la variable local a la función (parámetro formal).
- **Por referencia**: se pasa una referencia al contenedor, por tanto, NO se realiza la copia de sus elementos, pero sus valores pueden ser alterados dentro de la función.

Por tanto, la opción **más eficiente** es el **paso por referencia**, sobre todo cuando el contenedor contiene muchos elementos. Sólo en el caso que la función vaya a realizar una copia del contenedor original tendrá sentido pasar por valor.

Además, en el caso de que **no se vayan a modificar los elementos del contenedor** en la función, se puede utilizar el modificador ***const*** en el parámetro referenciado, que impide que sus elementos sean alterados.

Ejemplo:

```
bool minimo_vector_v4(const vector<int>& v, int& minimo)
```

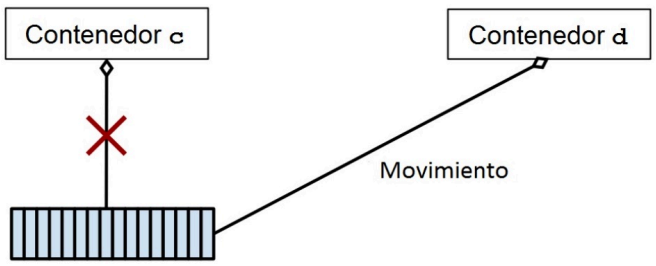
2. Contenedor como valor de retorno de una función

C++ permite devolver objetos como valor de retorno de una función, esto incluye a los contenedores.

Además, para garantizar la eficiencia, NO se realiza una copia de los elementos del contendor devuelto a la variable asignada en la llamada a la función, sino que se asocia directamente la zona de memoria del contenedor a dicha variable.

Ejemplo:

```
contenedor<T> d = func(...);
...
contenedor<T> func(...) {
  contenedor<T> c;
  ...
  return c;
}
```



3.1. Ejemplo: función minimo_vector()

Dos versiones de una función que calcula el mínimo de un vector:

Utilizando arrays estilo C

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

bool minimo_vector_v1(int [], int, int&);

int main()
{
  int v[]{4,1,5,3};
  int minimo;

  if (minimo_vector_v1(v, 4, minimo))
    cout << "El mínimo es " << minimo << endl;
}

bool minimo_vector_v1(int v[], int num, int& minimo)
{
  if (num <= 0)
    return false;

  minimo = v[0];
  for (int i = 1; i < num; ++i)
    if (v[i] < minimo)
      minimo = v[i];

  return true;
}
```

Utilizando el contenedor vector

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

bool minimo_vector_v4(const vector<int>& v, int& minimo);
int main()
{
  vector<int>v{4,1,5,3};
  int minimo;

  if (minimo_vector_v4(v, minimo))
    cout << "El mínimo es " << minimo << endl;
}

bool minimo_vector_v4(const vector<int>& v, int& minimo)
{
  if(v.empty())
    return false;

  // v.push_back(0);  //Descomenta esta línea ¿Qué ocurre?
  minimo = v[0];
  for (size_t i = 1; i < v.size(); ++i)
    if (v[i] < minimo)
      minimo = v[i];

  return true;
}
```

3.2. Ejemplo: función carga_vector()

Dos posibles opciones para cargar desde teclado los elementos de un vector:

- Pasando el vector por referencia
- Devolviendo vía `return` el vector

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

void carga_vector_v1(vector<double>&);
vector<double> carga_vector_v2(void);
void muestra_vector(const vector<double>&);

int main ()
{
  vector<double> v;
  carga_vector_v1(v);
  muestra_vector(v);
  v=carga_vector_v2();
  muestra_vector(v);
}

void muestra_vector(const vector<double>& v)
{
  for (size_t i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
}
```

v1. Vector por referencia

```
void carga_vector_v1(vector<double>& v)
{
  int num_elementos;
  cout << "Introduce número de elementos: ";

  cin >> num_elementos;

  v.clear(); //Borramos el vector por si v no está vacío
  for (int i = 0; i < num_elementos; ++i)
  {
    double valor;
    cout << "\nIntroduce elemento " << i+1 << ": ";
    cin >> valor;
    v.push_back(valor);
  }
}
```

v2. Vector devuelto vía return

```
vector<double> carga_vector_v2()
{
  int num_elementos;
  cout << "Introduce número de elementos: ";
  cin >> num_elementos;

  vector<double> v;
  v.reserve(num_elementos); // Creamos al menos num_elementos de capacidad

  for (int i = 0; i < num_elementos; ++i)
  {
    double valor;
    cout << "\nIntroduce elemento " << i+1 << ": ";
    cin >> valor;
    v.push_back(valor);
  }
  return v;
}
```

4. Iteradores

Definición

Un iterador es un componente del lenguaje que permite navegar o **recorrer secuencialmente los elementos de una colección**, en este caso de un contenedor.

Durante el recorrido, el iterador va referenciando o apuntando a cada uno de los elementos del contenedor, **permitiendo acceder y modificar su contenido**.

Los iteradores permiten recorrer **de forma unificada** cualquier tipo de contenedor, independientemente de su implementación, y si permite o no el uso de índices.

Formas de utilización

C++ permite utilizar iteradores de forma **explícita** o **implícita**.

4.1. Explícitos

1. Declaración

contenedor<T>::iterator identificador

Ejemplo:

```
vector<int>::iterator it  // iterador 'it' que permite recorrer un contenedor vector de enteros
```

2. Recorrido del contenedor

2.1. Métodos begin() y end()

Para recorrer un contenedor mediante un iterador es necesario hacer uso de los métodos:

```
begin()  // devuelve un iterador al primer elemento del contenedor
```

```
end()  // devuelve un iterador al último elemento
```

2.2. Operador *

Permite acceder al elemento referenciado por el iterador.

```
*identificador_iterador  // acceso al elemento apuntado
```

2.3. Ejemplo

// Uso de iteradores

```
#include <iostream>
#include <vector>

using namespace std;

main()
{
  vector<char> v;

  for (int x = 'A'; x <= 'Z'; x++)
    v.push_back(x);

  // Declaración del iterator
  vector<char>::iterator it;

  // Recorrido y visualización de los elementos del contenedor
  cout << "Elementos del contenedor:" << endl;
  for( it = v.begin(); it != v.end(); it++ )
    cout << *it << endl;

}
```

3. Otros tipos de iteradores

3.1. Iterador inverso

Permite recorrer el contenedor de forma inversa (desde el final hasta el inicio).

Declaración

```
contenedor<T>::reverse_iterator identificador
```

Métodos

```
rbegin()  // devuelve un iterador inverso al último elemento
```

```
rend()  // devuelve un iterador inverso al primer elemento
```

Ejemplo:

```
#include <iostream>
#include <list>

using namespace std;

main()
{
  list<char> v;  // lista de caracteres

  // Inicialización de la lista
  for (int x = 'A'; x <= 'Z'; x++)
    v.push_back(x);

  cout << "Orden original" << endl;
  list<char>::iterator i = v.begin();  // iterador (hacia adelante)
  while( i != v.end() )
  {
    cout << *i++ << " ";  // accede al elemento y luego incrementa
  }
  cout << endl;

  cout << "Orden inverso" << endl;
  list<char>::reverse_iterator ri = v.rbegin();  // iterador inverso
  while( ri != v.rend() )
  {
    cout << *ri++ << " ";
  }
  cout << endl;

}
```

3.2. Iterador constante

Tanto *iterator* como *reverse_iterator* son iteradores que permiten acceder y modificar el contenido de los elementos del contenedor.

En cambio, un iterador constante **permite acceder a un elemento pero no modificarlo**.

Declaración

```
contenedor<T>::const_iterator identificador
```

```
contenedor<T>::const_reverse_iterator identificador
```

Métodos

```
cbegin()  // devuelve un iterador constante al primer elemento
```

```
cend()    // devuelve un iterador constante al último elemento
```

```
crbegin() // devuelve un iterador constante inverso al primer elemento
```

```
crend()   // devuelve un iterador constante inverso al último elemento
```

Ejemplo:

```
#include <iostream>
#include <vector>
using namespace std;
main()
{
  vector<int> v{11, 22, 33, 44, 55, 66, 77};

  vector<int>::const_iterator c_it = v.cbegin();  // iterador constante
  while ( c_it != v.cend() )
  {
    cout << *c_it << " ";
    // *c_it = 100;  // Error, iterador no modificable
    c_it++;
  }
}
```

4. Funciones no miembro: begin() y end()

Devuelven el iterador inicial y final respectivamente para un contenedor pasado como parámetro.

Estas funciones son **compatibles con los arrays tipo C**.

Ejemplo:

```
#include <iostream>
using namespace std;
main()
{
  int v[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
  cout << "Contenido del array:" << endl;
  for (auto it = begin(v); it != end(v); ++it)
  {
    cout << *it << " ";
  }
}
```

4.2. Bucle for 'por rango'

C++ dispone de un bucle for especializado en recorrer todos los elementos de una colección, denominado *for por rango*.

1. Sintaxis

```
for (tipo_elemento id_elemento : id_contenedor)
```

id_elemento es un **iterador implícito** del tipo *tipo_elemento* que va tomando los valores de los elementos del contenedor que va recorriendo.

Esta variable contiene una copia del elemento del contenedor al que apunta en cada momento, por tanto, si se modifica el valor del iterador, el contenedor no se verá afectado.

```
for (tipo_elemento& id_elemento : id_contenedor)
```

En este caso, *id_elemento* es un **iterador implícito por referencia**, por tanto, una modificación del valor del iterador afecta al valor almacenado en el contendor.

2. Ejemplo: función suma_elementos()

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

int suma_elementos(const vector<int>&);

int main ()
{
    vector<int> v{1, 2, 3, 4, 5};
    int suma = suma_elementos(v);
    cout << "La suma de los elementos es " << suma << endl;
}

int suma_elementos(const vector<int>& v)
{
    int suma = 0;
    for (int x : v)
        suma += x;
    return suma;
}
```

3. for 'clásico' vs for 'por rango'

A continuación, se muestra un ejemplo de función que muestra por pantalla los valores de un vector.

La primera versión utiliza un for 'clásico', mientras que la segunda versión utiliza un for 'por rango':

```
void muestra_vector_v1(const vector<int>& v)
{
  for (size_t i = 0; i < v.size(); ++i)
    cout << v[i] << endl;
}

void muestra_vector_v2(const vector<int>& v)
{
  for (int x : v)
    cout << x << endl;
}
```

4.3. El especificador 'auto'

1. Utilización del especificador 'auto'

Declarar una variable inicializándola con el resultado de una expresión o con el valor devuelto por una función es una tarea recurrente en programación.

Ejemplo:

```
string fun();
...
int a =  2 + 3;
string b = fun();
```

En ocasiones, puede ser muy útil que el **compilador deduzca automáticamente el tipo** de una variable en función del tipo asociado a la expresión o el valor devuelto por una función con el que se inicializa. Para ello, basta con utilizar el especificador 'auto' en la declaración de dicha variable.

Ejemplo:

```
string fun();
...
auto a =  2 + 3;  // a se define como tipo int
auto b = fun();  // b se define como tipo string
```

2. Utilización con iteradores

La utilización del especificador 'auto' puede simplificar significativamente el código asociado al uso de iteradores, debido a la variabilidad de tipos que pueden manejar, y que habitualmente son definidos a partir del valor devuelto por un método.

Ejemplo:

```
void muestra_vector_v3(const vector<int>& v)
{
    for (auto x : v)
        cout << x << endl;
}
```

3. Ejemplo: función suma_elementos() utilizando 'for por rango' y 'auto'

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

int suma_elementos(const vector<int>&);
int main ()
{
    vector<int> v{1, 2, 3, 4, 5};  // ¿Se podría utilizar auto?
    auto suma = suma_elementos(v);
    cout << "La suma de los elementos es " << suma << endl;
}
int suma_elementos(const vector<int>& v)
{
    int suma = 0;  // ¿Se podría utilizar auto?
    for(auto x : v)
        suma += x;
    return suma;
}
```

5. Contenedores y métodos

En los siguientes apartados, se muestra un resumen de los principales contenedores y sus métodos, disponibles en la biblioteca estándar de C++.

5.1. queue (Cola)

queue<T>

1. Principales métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Propios		
push(elemento)	Inserta un elemento en la cola (al final).	P1: elemento
pop()	Elimina el primer elemento de la cola.	
Acceso a elementos	(Devuelven una referencia al elemento)	
front()	Acceso al primer elemento.	
back()	Acceso al último elemento.	
Tamaño		
empty()	Chequea si el contenedor está vacío.	R: true/false
size()	Devuelve el tamaño.	R: size_t

2. Otras herramientas

No dispone de:

- Acceso mediante índices
- Iteradores
- Operador =

3. Ejemplo

```
// queue::push/pop
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
  queue<int> myqueue;
  int myint;

  cout <<  "Please enter some integers (enter 0 to end):\n" ;

  do {
    cin >> myint;
    myqueue.push (myint);
  } while (myint);

  cout <<  "myqueue contains: " ;
  while (!myqueue.empty())
  {
    cout <<  ' '  << myqueue.front();
    myqueue.pop();
  }
  cout <<  '\n' ;

  return 0;
}
```

5.2. stack (Pila)

stack<T>

1. Principales métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Propios		
push(elemento)	Inserta un elemento en la parte superior de la pila.	P1: elemento
pop()	Elimina el elemento superior de la pila.	
top()	Acceso al elemento superior de la pila.	
Tamaño		
empty()	Chequea si el contenedor está vacío.	R: true/false
size()	Devuelve el tamaño.	R: size_t

2. Otras herramientas

No dispone de:

- Acceso mediante índices
- Iteradores
- Operador =

3. Ejemplo

```
// stack::push/pop
#include <iostream>
#include <stack>
using namespace std;

int main ()
{
  stack<int> mystack;

  for (int i=0; i<5; ++i) mystack.push(i);

  cout << "Popping out elements..." ;
  while (!mystack.empty())
  {
     cout << ' ' << mystack.top();
     mystack.pop();
  }
  cout << '\n' ;

  return 0;
}
```

5.3. list (Lista)

list<T>

1. Principales métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Modificadores		
push_front(elemento)	Inserta un elemento al comienzo de la lista.	P1: elemento<T>
pop_front()	Elimina el primer elemento.	
push_back(elemento)	Inserta un elemento al final de la lista.	P1: elemento<T>
pop_back()	Elimina el último elemento.	
clear()	Borra todo el contenido.	
assign(elementos)	Asigna un nuevo contenido al contenedor.	Constructor: Asignación y Relleno
Operaciones con listas		
remove(valor)	Elimina los elementos con un valor determinado.	P1: valor<T>
unique()	Elimina valores duplicados consecutivos de la lista.	
merge(lista)	Fusiona las listas de manera ordenada.	P1: lista<T> ordenada (Queda vacía)
sort()	Ordena los elementos de la lista.	
reverse()	Invierte el orden de los elementos.	
Acceso a elementos	(Devuelven una referencia al elemento)	
front()	Acceso al primer elemento.	
back()	Acceso al último elemento.	
Tamaño		
empty()	Chequea si el contenedor está vacío.	R: true/false
size()	Devuelve el número de elementos del contenedor.	R: size_t

2. Otras herramientas

Dispone de:

- Iteradores
- Operador =

No dispone de:

- Acceso mediante índices

3. Ejemplo

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

main()
{
  list<string> mylist;

  mylist.push_back ("one");
  mylist.push_back ("two");
  mylist.push_back ("three");

  cout << "Lista original:";
  for (auto x : mylist)
    cout << ' ' << x;
  cout << endl;

  mylist.sort();  // Se ordena la lista

  cout << "Lista ordenada:";
  for (auto x : mylist)
    cout << ' ' << x;
  cout << endl;

  return 0;
}
```

5.4. set (Conjunto)

set<T>

1. Principales métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Modificadores		
insert(elemento)	Inserta un elemento (o lista de elementos). (Los elementos existentes no son insertados)	P1: elemento<T>
erase(elemento)	Elimina un elemento determinado.	P1: elemento<T> R: 1: elemento eliminado, 0: no existe el elemento
clear()	Borra todo el contenido.	
Búsqueda		
count(elemento)	Busca un elemento determinado.	P1: elemento<T> R: 1: existe el elemento, 0: no existe el elemento
Tamaño		
empty()	Chequea si el contenedor está vacío.	R: true/false
size()	Devuelve el tamaño.	R: size_t

2. Otras herramientas

Dispone de:

- Iteradores
- Operador =

No dispone de:

- Acceso mediante índices

3. unordered_set (Conjunto no ordenado)

Versión no ordenada del contenedor set, la cual ofrece **inserciones y eliminaciones de elementos muy rápidas**.

unordered_set<T>

Métodos

Similares al contendor set.

Otras herramientas

Similares a set, salvo que sólo dispone de iteradores hacia delante.

4. Ejemplos

```
// unordered_set::count
#include <iostream>
#include <unordered_set>
using namespace std;
int main ()
{
  unordered_set<string> myset = { "hat", "umbrella", "suit" };
  for (auto x: {"hat","sunglasses","suit","t-shirt"}) {
    if (myset.count(x) > 0)
      cout << "myset has " << x << endl;
    else
      cout << "myset has no " << x << endl;
  }
  return 0;
}
```

```
// unordered_set::insert
#include <iostream>
#include <unordered_set>
using namespace std;
int main ()
{
  unordered_set<string> myset = {"yellow","green","blue"};
  string mystring = "red";
  myset.insert (mystring);
  myset.insert ( {"purple","orange"} );
  cout << "myset contains:";
  for (const string& x: myset)
    cout << " " << x;
  cout << endl;
  return 0;
}
```

5.5. map (Conjunto clave-valor)

map<keyT, T>

1. Principales métodos

MÉTODO	DESCRIPCIÓN	PARÁMETROS
Modificadores		
insert(elemento)	Inserta un elemento (o lista de elementos). (Los elementos existentes no son insertados)	P1: elemento<keyT, T>
erase(clave)	Elimina el elemento con una clave determinada.	P1: clave<keyT> R: 1: elemento eliminado, 0: no existe el elemento
clear()	Borra todo el contenido.	
Búsqueda		
count(clave)	Busca el elemento con una clave determinada.	P1: clave<keyT> R: 1: existe el elemento, 0: no existe el elemento
Tamaño		
empty()	Chequea si el contenedor está vacío.	R: true/false
size()	Devuelve el tamaño.	R: size_t

2. Otras herramientas

Dispone de:

- Iteradores
- Operador =
- **Acceso mediante clave**: operador[] y at()

3. unordered_map (Conjunto clave-valor no ordenado)

Versión no ordenada del contenedor map, la cual ofrece **inserciones y eliminaciones de elementos muy rápidas**.

unordered_map<keyT, T>

Métodos

Similares al contendor map.

Otras herramientas

Similares a map, salvo que sólo dispone de iteradores hacia delante.

4. Ejemplos

```
// map::insert
#include <iostream>
#include <map>
using namespace std;
int main ()
{
  map<char,int> mymap;
  mymap.insert ( { {'b',1}, {'a',2} } );  // Inserta lista de elementos (clave-valor)
  mymap['z'] = 100;  // Si no existe, crea el elemento
  // mymap.at('t') = 200;  // Error, fuera de rango
  mymap.at('a') = 5;  // Modifica su valor
  cout << "mymap contains:\n";
  for (auto x: mymap)
    cout << x.first << "-" << x.second << endl;  // Acceso a la clave (.first) y valor (.second) de cada elemento

  return 0;
}
```

Reiniciar tour para usuario en esta página