C++ Standard Template Library

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Introdução

O que é a STL?

A STL (Standard Template Library) é um conjunto de classes e funções "template" para:

- "containers" - estruturas de dados que permitem albergar coleções elementos de várias formas;
- iteradores - permitem percorrer "containers" e outras classes;
- algoritmos - para operações comuns por ex. como pesquisa e ordenação;

Documentação da STL

- "Containers library"
- "Iterators library"
- "Algorithms library"

Tipos de "container"

A STL disponibiliza vários tipos de "containers" na forma de classes template:

- **Sequências de elementos** ("sequence containers"): `vector`, `list`, `deque` ...
- **Associação de elementos** ("associative containers"): `map`, `set`, `multiset`, `multimap`
- **"Container adapters"**: definem estruturas de dados comuns, usando outros "containers" internamente para guardar os elementos: `stack`, `queue`, ...
- Outras classes, não sendo "containers", têm funcionalidade similar. Um exemplo é a classe `std::string` que funciona como "container" de valores `char`.

Vantagens e desvantagens

+ Implementação de estruturas de dados comuns e algoritmos associados. Programadores não têm de "re-inventar a roda", e em particular lidar com uso de memória dinâmica e apontadores (usados internamente por "containers" STL).

− Desenho algo irregular; por ex. "containers" com funcionalidade aproximada têm interface diferente sem razão aparente (ex. `list` e `vector`). Semelhanças entre "containers" não capturadas por uma hierarquia de classes. Mensagens de erro durante compilação são frequentemente difíceis de interpretar.

+ / − Containers guardam elementos por cópia, o que evita confusões com apontadores mas por outro lado pode levar a cópias desnecessárias. Por sua vez, se os elementos forem apontadores, têm de ser geridos pelo programador de forma externa a um "container" ou usando "smart points" por ex. usando `std::auto_ptr`.

Containers - Operações comuns

Função	Descrição
Construtor por omissão	Cria "container" vazio.
Construtor por cópia	Cria "container" a partir de outro, copiando elementos.
Construtor via "initializer list"	Cria "container" a partir de elementos em "initializer list" (desde C++ 11).
Destrutor	Liberta recursos associados ao "container" quando este deixa de ser usado.
`size`	Devolve número de elementos.
`empty`	Indica se "container" está vazio.
`operator=`	Operador de atribuição.
`operator==`	Operador de comparação.

Containers - Operações comuns (cont.)

Função	Descrição
`at`	Acede a elemento.
`insert`	Inserção de elementos.
`erase`	Remoção de elementos.
`clear`	Apaga todos os elementos.
`begin`	Devolve iterador para o início.
`end`	Devolve iterador assinalando fim.
`rbegin`	Devolve iterador para iteração reversa.
`rend`	Devolve iterador assinalando fim de iteração reversa.

“Sequence containers”

Implementações

Classe	Tipo de implementação
`vector`	Array dinâmico - elementos armazenados contiguamente num array cujo tamanho é dinamicamente ajustado.
`list`	Lista duplamente ligada - elementos armazenados em lista de nós duplamente ligados (i.e., com apontador para nós anterior e próximo).
`forward_list`	Lista simplesmente ligada - elementos armazenados em lista de nós simplesmente ligados (i.e., apenas com apontador para próximo nó).
`deque`	Vector de vectores (esquema de dupla indireção).

Operações sobre início e fim

Função	Descrição
`front`	Acesso ao primeiro elemento.
`back`	Acesso ao último elemento (não definida para `forward_list`).
`push_back`	Adiciona elemento ao fim da sequência (não definida para `forward_list`).
`pop_back`	Remove elemento do fim da sequência (não definida para `forward_list`).
`push_front`	Adiciona elemento ao início da sequência (não definida para `vector`).
`pop_front`	Remove elemento do início da sequência (não definida para `vector`).

Complexidade algorítmica

Operação	`vector`	`list`[1]	`deque`
`pop_front` e `push_front`	N/A	$O(1)$	$O(1)$
`pop_back` e `push_back`	$O(1)$	$O(1)$	$O(1)$
`at` (indexação)	$O(1)$	$O(n)$	$O(1)$
`insert`/`remove` [2]	$O(n)$	$O(1)$	$O(n)$

[1] `forward_list` similar mas não define `pop_back` e `push_back`.

[2] `insert` e `remove` via iterador p/posição.

$O(1)$ - **tempo constante** (ou constante amortizado em alguns casos) : esforço computacional é independente do número de elementos no "container". Tempo de realocação, se necessário (como em `vector`), é linear mas amortizado em chamadas subsequentes.

$O(n)$ - **tempo linear** : o esforço computacional é proporcional ao número de elementos (n) no "container".

Exemplo

Já vimos vários exemplos com `vector`. O uso de `list` ou `deque` é análogo no que respeita às funções membro mais elementares:

```
#include <deque>
....
std::deque<std::string> dq {"a", "b", "c", "d", "e"};
dq.push_back("f");
dq.push_front("g");

std::cout << dq.size() << " [";
for (size_t i = 0; i < dq.size(); i++)
  std::cout << ' ' << dq.at(i);
std::cout << " ]" << std::endl;
```

Output:

```
7 [g a b c d e f]
```

Construção "in-place" (C++ 11)

Operações `push_back`/`push_front` (entre outras) podem requerer criação de objectos temporário e sua cópia. Desde o C++ 11, funções membro como `emplace_front` ou `emplace_back` permitem passar argumentos ao, construtor da classe para criação "in-place".

```
class coord2d {
public:
  int x, y;
  coord2d(int x, int y) : x(x), y(y) { }
};
std::list<coord2d> list;
list.push_back(coord2d(1,2)); // construtor invocado
list.push_back({ 1,2 }); // implicitamente equivalente
list.emplace_back(1,2); // construção in-place
```

Iteradores

O que são?

- Um iterador é um objecto que permite iterar elementos de um "container".
- Cada iterador tem associada uma noção de posição dentro do "container" a que se refere e ao elemento correspondente se a posição for válida.
- Iteradores são usados para iterar um "container", mas também em associação a funções membro do "container" (ex. `erase` e `insert`) ou algoritmos da STL (ex. `std::sort`).

Exemplo

Cálculo da soma de um vector de inteiros:

```
std::vector<int> c { 2, -5, 6, 7, -9 };
int sum = 0;
for( std::vector<int>::iterator itr = c.begin();
     itr != c.end();
     itr++) {
   sum += *itr;
}
std::cout << sum << std::endl;
```

- Iterador inicial obtido com `c.begin()`.
- Detetamos fim da iteração comparando `itr` com `c.end()` (iterador sem elemento associado que "marca" o fim).
- Elemento associado ao iterador obtido com operador de dererenciação: `*itr`.
- Iterador avança para próxima posição com `itr++`.

Tipos de iterador

- **"Forward iterators"**: apenas permitem avançar para a frente uma posição;
  - ex. iteradores de `forward_list`
- **Bi-direccionais** - também permitem avançar para trás
  - ex. iteradores de `list`
- **Acesso aleatório ("random acess")**: permitem andar para trás ou para a frente um número arbitrário de elementos
  - ex. iteradores de `vector`, `queue`

Operadores definidos para `iterator`

Operadores	Significado
`*itr`	Obtém referência para elemento.
`itr1 = itr2`	Atribuição
`itr1 == itr2`	Comparação
`itr1 != itr2`	
`itr1 < itr2`	
`++itr`	Avança para próxima posição.
`itr++`	
`--itr`	Recua para posição anterior (it. bi-direcionais)
`itr--`	
`itr += n`	Saltos de `n` posições (“random access iterators”).
`itr -= n`	
`itr + n`	
`itr - n`	

Exemplo (cont.)

No exemplo anterior é mais conveniente empregar `auto` para para evitar declarar o tipo (tipicamente verboso) do iterador, isto é:

```
std::vector<int> c { 2, -5, 6, 7, -9 };
int sum = 0;
for( auto itr = c.begin();
     itr != c.end();
     itr++) {
   sum += *itr;
}
std::cout << sum << std::endl;
```

Iteradores para modificação

Podemos também empregar iteradores para modificar o conteúdo de um "container".

Exemplo - incremento de todos os valores de uma vector de inteiros:

```
std::vector<int> c { 2, -5, 6, 7, -9 };
for( auto itr = c.begin();
     itr != c.end();
     itr++) {
   *itr = *itr + 1;
}
std::cout << sum << std::endl;
// valores no final: 3, -4, 7, 8, -8
```

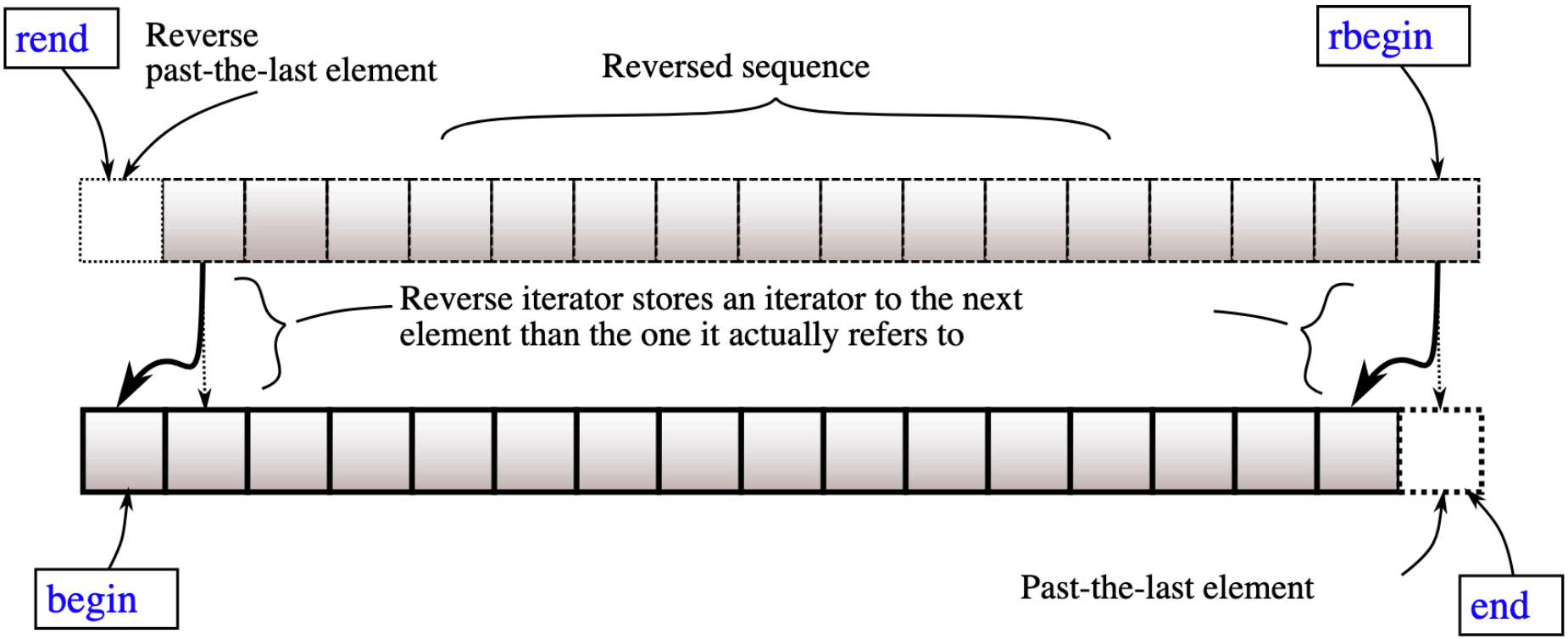
Iteradores reversos

Para “containers” podemos também usar iteradores “reversos”. Estes permitem iterar elementos na ordem inversa de um iterador standard. As funções membro `rbegin()` e `rend()` devem ser empregues para esse propósito.

```
for( auto itr = c.rbegin();
     itr != c.rend();
     itr++) {
   ...
}
```

Iteradores reversos (cont.)

Ilustração para `std::vector` (imagem de cplusplus.com):



Ciclos “for-each” e iteradores

Em um ciclo “for-each” com um “container” está **subjacente o uso implícito de um iterador**.

Por exemplo

```
for( int& v : c) {
  v = v + 1;
}
```

é equivalente a

```
for( auto itr = c.begin();
     itr != c.end();
     itr++) {
  *itr = *itr + 1;
}
```

Iteração com índices vs uso de iteradores

Se `c` é do tipo `std::vector` então a iteração

```
for( auto itr = c.begin(); itr != c.end(); itr++) {
  sum += *itr;
}
```

tem o mesmo efeito lógico e também a mesma eficiência do que

```
for( auto i = 0;  i < c.size(); i++) {
  sum += c.at(i);
}
```

porque `at` tem tempo de execução constante.

Mas **tipicamente não é assim e devemos caso geral empregar iteradores**. Por exemplo se `c` for do tipo `std::list` então `at` tem tempo de execução linear, enquanto que o uso do iterador com `*itr` tem tempo de execução constante.

Inserções e remoções

Em "containers" de sequência, podemos usar iteradores para inserir e remover elementos, resp. via `erase` ou `insert`:

```
std::vector<int> c;
...
// Remove 3º elemento
c.erase(itr.begin() + 2);
// Insere 123 na penúltima posição
c.insert(itr.end() - 2, 123);
```

Inserções e remoções (cont.)

... e isso pode acontecer durante a iteração:

```
std::list<int> c { 2, 0, 1, 0, 2 };
auto itr = c.begin();
while (itr != c.end()) {
  if (*itr == 0) itr = c.erase(itr);
  else itr++;
}
// No final c tem elementos: 2, 1, 2
```

`insert` e `erase` tornam inválido o iterador dado ou outros que possam existir sobre o "container", mas retornam um novo que o é (em `erase` acima: o da posição a seguir).

Algoritmos STL

Perspectiva geral

Algoritmos STL:

- conjunto de funções template definidas no header `algorithm`
- implementam funcionalidades comuns como ordenação, pesquisa, cópia de elementos, preenchimento de containers ...
- tipicamente tomam como argumentos iteradores e não "containers" directamente;

Exemplos de operações

Funções	Descrição
`sort`	Ordenação
`search`	Pesquisa linear
`binary_search`	Pesquisa binária
`for_each`	Iteração
`transform`	Transformação de valores
`unique`	Remoção de duplicados
`min`, `max`	Obtenção de mínimo e máximo
...	... várias outras ...

Para detalhes veja documentação online.

Vamos ilustrar a seguir o uso de `std::sort`.

Ordenação

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

`std::sort` toma como argumentos dois iteradores resp. para o início e fim dos elementos a iterar. Primeira variante ordena elementos de acordo com operador `<` definido para `T`, e segunda variante toma um argumento para especificar a função de comparação.

Os iteradores têm de ser do tipo "random access", ex. iteradores de um `vector` ou `deque`; para `list` podemos usar a função membro `sort` (que não existe para `vector` e `deque`).

Exemplos de ordenação

Função de comparação deverá ter 2 argumentos `a` e `b`do tipo usado com a função "template" e retornar um valor `bool` indicando se `a` deve preceder `b`.

```
bool increasing_order(int a, int b)
  { return a < b; }
bool decreasing_order(int a, int b)
  {  return a > b; }
...
std::vector v { 1, 7, 3, -1, 9, 1, 10 };
std::sort(v.begin(), v.end()); // (1)
std::sort(v.begin(), v.end(), decreasing_order); // (2)
std::sort(v.begin(), v.end(), increasing_order); // (3)
```

Exemplos de ordenação (cont.)

Exemplo anterior:

```
bool increasing_order(int a, int b)
  { return a < b; }
bool decreasing_order(int a, int b)
  {  return a > b; }
...
std::vector v { 1, 7, 3, -1, 9, 1, 10 };
std::sort(v.begin(), v.end()); // (1)
std::sort(v.begin(), v.end(), decreasing_order); // (2)
std::sort(v.begin(), v.end(), increasing_order); // (3) = (1)
```

Ordenação obtida em cada caso:

```
(1) [ -1 1 1 3 7 9 10 ]
(2) [ 10 9 7 3 1 1 -1 ]
(3) [ -1 1 1 3 7 9 10 ]
```

Exemplos de ordenação (cont.)

```
struct Date { int day, month, year; };

bool by_year(const Date& a, const Date& b) {
    return a.year < b.year;
}
...
std::vector<Date> vec { ... };
std::sort(vec.begin(), vec.end(), by_year);
```

Sem ordenação:

```
[ 1/7/2021 30/12/2020 30/11/2020 14/5/2020
  14/5/2021 12/5/2020 13/5/2020 ]
```

Ordenação com `by_year`:

```
[ 30/12/2020 30/11/2020 14/5/2020 12/5/2020
  13/5/2020 1/7/2021 14/5/2021 ]
```

Exemplos de ordenação (cont.)

```
// Ordenação crescente por ano, depois por mês,
// e finalmente por dia.
bool by_year_month_and_day(const Date& a, const Date& b) {
    if (a.year != b.year) return a.year < b.year;
    if (a.month != b.month) return a.month < b.month;
    return a.day < b.day;
}
```

Sem ordenação:

```
[ 1/7/2021 30/12/2020 30/11/2020 14/5/2020
  14/5/2021 12/5/2020 13/5/2020 ]
```

Ordenação com `by_year_month_and_day`:

```
[ 12/5/2020 13/5/2020 14/5/2020 30/11/2020
  30/12/2020 14/5/2021 1/7/2021 ]
```

“Associative containers” (conjuntos e mapas)

Tipos de "containers"

A STL disponibiliza implementações para:

- **Conjuntos** - `set`, `unordered_set`
- **Mapas** (também chamados de dicionários): conjunto de pares chave-valor em que chaves são valores únicos - `map`, `unordered_map`
- **Multi-conjuntos e multi-mapas** - conjuntos/mapas com repetição de elementos/chaves - `multi_set`, `multi_map`, `unordered_multi_set`, `unordered_multi_map`

Implementações:

`set`, `map`, `multi_map`, `multi_set`

- elementos **têm** ordem associada;
- implementação baseada em **árvores de pesquisa binária "Red-Black"**
- tempo de pesquisa, inserção e remoção de elementos é **logarítmico** - $O(\log n)$

`unordered_set`, `unordered_map`, `unordered_multi_map`, `unordered_multi_set`

- elementos **não** têm ordem associada;
- implementação baseada em **tabela de dispersão ("hash table")**
- tempo de pesquisa, inserção e remoção de elementos é **constante** - $O(1)$

Conjuntos - funções elementares

`set`, `unordered_set`

Operação	Descrição
`insert(v)`	Insere elemento `v` (sem efeito se `v` já está no conjunto).
`erase(v)`	Remove elemento `v` (sem efeito se `v` não está no conjunto).
`find(v)`	Verifica se elemento `v` está no conjunto (devolve iterador).

Conjuntos - exemplo de uso

```
std::set<int> s; // ou std::unordered_set
s.insert(1);
s.insert(2);
s.insert(3);
s.erase(2);
for (int i = 0; i <= 3; i++)
  if (s.find(i) != s.end())
    std::cout << i << " is in the set" << std::endl;
  else
    std::cout << i << " is not in the set" << std::endl;
```

```
0 is not in the set
1 is in the set
2 is not in the set
3 is in the set
```

set vs. unordered_set

set garante iteração ordenada, unordered_set não.

```
std::set<std::string> a
  {  "fcup", "prog", "feup", "leic" };
for (auto v : a ) { std::cout << v << ' ';
...
std::unordered_set<std::string> b
  {  "fcup", "prog", "feup", "leic" };
for (auto v : a ) { std::cout << v << ' ';
```

dá-nos

```
fcup feup leic prog
leic feup prog fcup
```

`set` vs. `unordered_set` (cont.)

- `set` requer que operador `<` esteja definido para o tipo usado com a template
  - dois elementos `a` e `b` são considerados equivalentes se `! (a < b) && ! (b < a)`
  - função de comparação pode ser dada via construtor em alternativa
- `unordered_set` requer para o tipo em questão que operador `==` esteja definido para testar igualdade de elementos e ainda função de "hashing"
  - para tipos primitivos e classes como `std::string` suporte para "hashing" já é dado via `std::hash`

Mapas

`map`, `unordered_map`

Operação	Descrição
`insert({k,v})`	Insere par `(k,v)` (sem efeito se já existe par com chave `k`).
`erase(k)`	Remove par com chave `k` (sem efeito se chave `k` não existe no mapa)
`find(k)`	Verifica se entrada com chave `k` existe (devolve iterador)
`at(k)`	Obtém valor para chave `k` (que deve existir).
`operator[](k)`	Obtém referência para valor para chave `k`, criando associação caso não exista.

Mapas - exemplo

```
std::map<std::string, int> m;
m.insert({"a", 1});
m.insert(std::pair<std::string,int>("b", 2));
m.insert({"a", 3}); // sem efeito
m.at("a") ++; // actualização
m.erase("b"); // remoção
m["c"] = 4; // inserção implícita
m["c"] += m["a"]; // actualização
```

dá-nos no final no mapa os pares chave-valor `(a,2)` e `(c,6)` (chave `b` é definida mas depois removida).

`std::pair<std::string,int>("b",2)` é equivalente a escrevermos `{"b",2}` em C++ 11 - veja documentação de `std::pair`.

Mapas - exemplo (cont.)

Exemplo anterior passo a passo

Passo	Conteúdo do mapa
Construção	`{ }` (vazio)
`m.insert({"a", 1})`	`{ (a,1) }`
`m.insert({"b", 2})`	`{ (a,1), (b,2) }`
`m.insert({"a", 3})`	`{ (a,1), (b,2) }` (inalterado)
`m.at("a") ++`	`{ (a,2), (b,2) }`
`m.erase("b")`	`{ (a,2) }`
`m["c"] = 4`	`{ (a,2), (c,4) }`
`m["c"] += m["a"]`	`{ (a,2), (c,6) }`

Mapas - exemplo (cont.)

Um mapa pode ser iterados para acedermos aos pares chave-valor contidos representados via `std::pair`.

Em continuação do exemplo anterior, o seguinte código

```
for (auto itr = m.begin(); itr != m.end(); itr++)
    std::cout << (*itr).first << " --> "
              << (*itr).second << std::endl;
```

ou então, de forma equivalente e mais simples (ciclo "for-each")

```
for (auto kv: m)
    std::cout << kv.first << " --> "
              << kv.second << std::endl;
```

dão-nos o seguinte output:

```
a --> 2
c --> 6
```

`map` vs. `unordered_map`

Diferenças análogas às existentes entre `set` e `unordered_set`:

- `map` garante iteração ordenada, `unordered_map` não.
- `map` requer operador `<` definido para o tipo das chaves ou função de comparação, e `unordered_map` requer definição do operador `==` e da função de "hashing" para o tipo das chaves.

Multi-conjuntos

`multiset` e `unordered_multiset` suportam multi-conjuntos, em que podemos ver várias instâncias do mesmo elemento. Funções membro são similares às de `set` e `unordered_set`. A função membro `count` retorna o número de instâncias para um dado elemento.

```
std::multiset<int> ms;
for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= i; j++) ms.insert(i);
for (auto v : ms)
    std::cout << v << " count: " << ms.count(v) << std::endl;
```

Output:

```
1 count: 1
2 count: 2
2 count: 2
3 count: 3
3 count: 3
3 count: 3
```

Multi-mapas

`multimap` e `unordered_multimap` suportam coleções de pares chaves-valores em que a mesma chave se pode repetir várias vezes.

```
std::multimap<int,int> ms;
for (int i = 1; i <=3; i++)
    for (int j = 1; j <= i; j++) ms.insert({i, i * 10 + j});
for (auto v : ms)
    std::cout << v.first << " --> " << v.second
              <<  " (count: "<< ms.count(v.first)
              << ")" << std::endl;
```

Output:

```
1 --> 11 (count: 1)
2 --> 21 (count: 2)
2 --> 22 (count: 2)
3 --> 31 (count: 3)
3 --> 32 (count: 3)
3 --> 33 (count: 3)
```