

Ch. 6 - Conteneurs standards

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



13 novembre 2020

Table des matières

- 1 Introduction
- 2 Itérateurs
- 3 Conteneurs séquentiels
- 4 Adaptateurs
- 5 Conteneurs associatifs

Introduction

Historique

- En C, il n'existe pas de structure de donnée autre que les tableaux

- `int t[10];`

Inconvénients

- Efficacité d'ajout médiocre
- Pas de contrôle de bornes
- Les programmeurs étaient amenés à implémenter leurs propres structures de données
 - Parfois très complexe
- En C++, la librairie standard fournit des *conteneurs*
- Plusieurs conteneurs existent, aux caractéristiques et performances variées

Types de conteneurs

■ Il existe deux types de conteneurs

- 1 Séquentiels : les données sont ordonnées en séquence et parcourues dans cet ordre
- 2 Associatifs : les données sont associées à des « clés » et parcourues dans cet ordre

```
vector<double> v;  
...
```

0	2.3
1	3.14
2	2.0
3	5
4	-1.1
5	6
6	23
7	2.42

```
map<double> m;  
...
```

abs	"romeo42"
aro	"beta23"
clr	"alpha33"
fpl	"tango47"
jlc	"charlie44"
mcd	"omega3"
nvs	"whisky17"
pbt	"password"

Conteneurs séquentiels

- Les données sont parcourues selon l'ordre de rangement
- Cet ordre dépend de la logique du conteneur
 - Ordre d'ajout,
 - Ordre croissant, etc.
- Pour accéder à un élément, il est nécessaire de spécifier un emplacement mémoire
 - À la fin
 - En troisième position
 - Après un autre élément

Exemple de conteneurs séquentiels

- Tableau : `array` et `vector`
- Liste doublement chaînée : `list`
- Liste chaînée de tableaux de taille fixé (compromis) : `deque`

Conteneurs associatifs

- Souvent mis en œuvre à l'aide de tableaux associatifs et de fonctions de hachage
 - Tableau associatif : les clés sont triées selon leur ordre « naturel »
 - Fonction de hachage : les clés sont triées selon la sortie de la fonction de hachage
- L'ajout et l'accès nécessitent une clé, pas un emplacement mémoire
 - L'élément associé à `abs` est `"romeo42"`

Exemple de conteneurs associatifs

- Tableau de associatif : `map`
- Table de hachage : `unordered_map`
- Ensembles : `set`, `unordered_set`

Opérations courantes

- Ajout d'un élément
- Suppression d'un élément
- Contrôle de bornes
- Itération
- Algorithmique
 - Tri de conteneurs séquentiels
 - Fusion de séquences triées
 - Recherche d'un élément
 - Calcul du maximum
 - Comparaisons lexicographiques
- Application de fonctions à tous les éléments

Construction, copie et destruction

- Construire un conteneur d'objets entraîne pour chacun de ses éléments soit
 - l'appel du constructeur par défaut
 - l'appel du constructeur de copie

Exemple

- `vector<point> v(3);` : constructeur par défaut de `point`
- `vector<point> w(v);` : constructeur de copie de `vector<point>` qui appelle le constructeur de copie de `point`
- La destruction d'un conteneur entraîne la destruction de ses éléments
 - Si allocation dynamique : destruction « manuelle » nécessaire

Remarques

- Chaque conteneur a ses propres performances
 - Ajout, suppression, recherche
- La complexité algorithmique est précisée dans la documentation
- Il faut donc choisir *judicieusement* son conteneur en fonction des besoins
- Les conteneurs sont amenés à allouer des espaces mémoires
 - Spécifié par un allocateur
 - Par défaut, les données sont souvent allouées dynamiquement
- Passer un conteneur par valeur (paramètre et retour) entraîne des copies
 - Utiliser le passage par référence ou par adresse
 - Utiliser le passage par sémantique de mouvement (cf. Ch. 9)

Construction, copie, affectation et destruction (2/2)

- La destruction d'un conteneur entraîne la destruction de ses éléments
 - Si allocation dynamique : destruction « manuelle » nécessaire

Remarque

- Prenez des précautions lors de l'utilisation de conteneurs
 - Temps, mémoire
 - Allocations dynamiques

Exemple 1

■ Fichier cont-intro.cpp

```
1  class A
2  {
3      int i;
4
5      public:
6          A(int i = 0) : i(i) { cout << "Building_" << this << endl; }
7
8          A(const A& a) : i(a.i) { cout << "Copying_" << &a << "_into_" << this << endl; }
9
10         ~A() { cout << "Destroying_" << this << endl; }
11
12         A& operator =(const A& a) //to track affectation
13         {
14             cout << "Affecting_" << &a << "_into_" << this << endl;
15             i = a.i;
16             return *this;
17         }
18     };
19
20 void f(vector<A> v) { cout << "Entering_f" << endl; }
```

Exemple 1

■ Fichier `cont-intro.cpp`

```
1  int main()
2  {
3      vector<A> v(3); cout << endl;
4
5      vector<A> w(v); cout << endl;
6
7      f(v); //passed by value
8      cout << endl;
9
10     vector<A*> y;
11     for(int i = 0; i < 4; i++)
12         y.push_back(new A(i));
13     cout << endl;
14
15     //memory leak here
16 }
```

Exemple 2

■ Fichier `cstr-def.cpp`

```
1  class point
2  {
3      int x, y;
4
5      public:
6          point(int x, int y) : x(x), y(y)
7              {
8                  cout << "Building_" << x << "," << y << " " << endl;
9              }
10 };
11
12 int main()
13 {
14     vector<point> v(3);
15 }
```

Itérateurs

Généralités

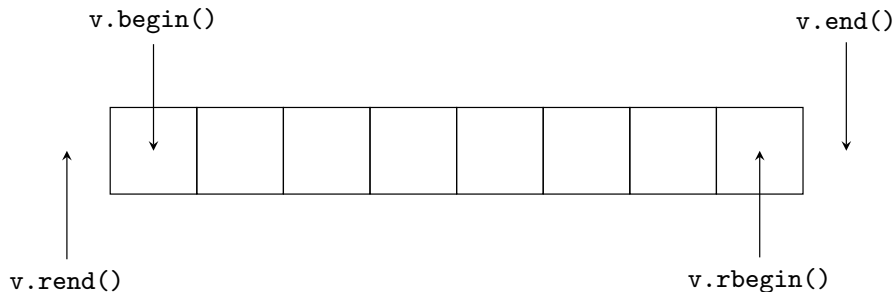
- Objets permettant de parcourir le contenu d'un conteneur
 - Boucle « `foreach` », etc.
- Utilisé régulièrement par la librairie sous la forme d'intervalle
 - Exemple : recherche d'un élément dans une partie du conteneur
- Existe sur tous les conteneurs
- Accès aux données via déréférencement

Deux types de parcours

- 1 Parcours « standard » : obtenus via `begin()` et `end()`
- 2 Parcours inverse : obtenus via `rbegin()` et `rend()`

Initialisation : schéma

```
vector<int> v(8);
```



Parcours : exemple

■ Fichier `iterate.cpp`

```
1  int main()
2  {
3      vector<int> v;
4      v.push_back(1); v.push_back(2); v.push_back(3);
5      vector<int>::iterator i;
6      for(i = v.begin(); i != v.end(); i++)
7          cout << *i << " ";
8      cout << endl;
9
10     for(int elem : v)
11         cout << elem << " ";
12     cout << endl;
13 }
```

Intervalle d'itérateurs

- Permet de spécifier un intervalle d'itérateurs pour un algorithme
- Toujours de la forme $[i, j[$
 - Pratique au vu de l'initialisation

Exemple

- Chercher un élément dans `[v.begin(), v.end() [`
- On suppose que j soit « atteignable » à partir de i
- Très utilisé dans `algorithm.h`

Exemple

■ Fichier `alg.cpp`

```
1  int main()
2  {
3      vector<int> v = {5,6,3,3,4,1,2};
4      list<int> l;
5
6      copy(v.begin(), v.end(), back_inserter(l));
7      cout << "list_=";
8      for(int e : l)
9          cout << e << " ";
10     cout << endl << endl;
11
12     equal(v.begin(), v.end(), l.begin()) ? cout << "true" : cout << "false";
13
14     cout << count(v.begin(), v.end(), 3) << endl;
15
16     auto i = find(v.begin(), v.end(), 3);
17     cout << *i << endl;
18
19     sort(v.begin(), v.end());
20     for(int e : v)
21         cout << e << " ";
22
23     for_each(v.begin(), v.end(), [](int i) { cout << "Hello_" << i << endl; } );
24 }
```

Conteneurs séquentiels

Introduction

- Ordonnés explicitement par le programme
 - `v[1]` dénote le deuxième élément de `v`
- Différent des conteneurs associatifs
 - Triés intrinsèquement par les clés ou une fonction de hachage
- Principaux représentants
 - 1 `array<T, N>` : tableau de taille `N` fixe
 - 2 `vector<T>` : tableau de taille dynamique
 - 3 `list<T>` : liste doublement chaînée
 - 4 `deque<T>` : liste doublement chaînée de tableaux de tailles fixes
- Performance des opérations varie en fonction du type de conteneur

Conclusion

- Choisir son conteneur en fonction de ses besoins

Construction

- Trois types de constructeurs de base pour `vector`, `list` et `deque`
 - 1 Vide (par défaut)
 - 2 Nombre d'éléments (identiques) donnés
 - 3 Intervalle d'itérateurs
- Pour `array`, seule la liste d'initialisation est autorisée

Exemple

```
1 array<int,4> a = {3, 1, 2, 5};  
2 vector<int> v;  
3 list<int> l(5);  
   ■ Pas list<int> l = 5; (constructeur explicit)  
   ■ deque<point> dq (5, point(2,1));  
4 vector<int> v(l.begin(), l.end());
```

Affectation, recopie et destruction

- L'implémentation des conteneurs passe souvent par une allocation dynamique de mémoire
 - Pas `array`
- Ils surchargent l'affectation et implémentent la recopie et destruction convenablement
- Pour l'affectation et la recopie : le type *doit* être identique
 - Y compris dans les paramètres templates

Exemple

- `vector<int> vi1 (...), vi2 (...);`
- `vector<double> vd1 (...);`
- `vi1 = vi2; //ok` `vd1 = vi1; //ko`

Autres fonctions globales

- `swap` échange le contenu de deux conteneurs
 - `v1.swap(v2);`
 - Plus efficace qu'un `swap` manuel
- `clear`
 - Taille nulle après `v.clear();` (pas avec `array`)
 - Appelle les destructeurs
- Opérateur `==` et `<`
 - Comparaison lexicographique, éléments comparés 2 à 2
 - Les objets doivent surcharger `==` ou `<`

Exemple

■ Fichier other.cpp

```
1  int main()
2  {
3      vector<int> v1 = {1,2,3,6};
4      vector<int> v2 = {2,3,4,5,0};
5
6      cout << ((v1 < v2) ? "v1_<v2" : "v2_>=v1") << endl;
7
8      v1.swap(v2);
9      for_each(v1.begin(), v1.end(), [](int& i) { cout << i << " "; });
10     cout << endl;
11     for_each(v2.begin(), v2.end(), [](int& i) { cout << i << " "; });
12     cout << endl;
13
14     point p1(1,1);
15     point p2(2,2);
16     vector<point> vp1; vp1.push_back(p1); vp1.push_back(p2);
17     vector<point> vp2; vp2.push_back(p1); vp2.push_back(p2);
18     cout << ((vp1 == vp2) ? "vp1_==vp2" : "vp1_!=vp2") << endl;
19
20     vp1.clear();
21 }
```

Insertion et suppression

- `vector`, `list` et `deque` offrent la possibilité d'ajouter et supprimer des éléments
 - Au début, à la fin ou sur une position arbitraire
 - Fonctions `push_back`, `pop_back`, `push_front`, `pop_front`, `insert`, `erase`
- Performances varient fortement en fonction du type de conteneur et de l'emplacement
- Ajout
 - `vector` : $\mathcal{O}(1)$ à la fin « s'il reste de la place », $\mathcal{O}(n)$ sinon
 - `deque` : $\mathcal{O}(1)$ au début ou à la fin « s'il reste de la place », $\mathcal{O}(n)$ sinon
 - `list` : $\mathcal{O}(1)$ au début, à la fin et « sur un itérateur »
- Suppression
 - `vector` : $\mathcal{O}(1)$ à la fin, $\mathcal{O}(n)$ sinon
 - `deque` : $\mathcal{O}(1)$ au début ou à la fin, $\mathcal{O}(n)$ sinon
 - `list` : $\mathcal{O}(1)$ au début, à la fin et « sur un itérateur »

Exemple

■ Fichier insert-erase.cpp

```
1  int main()
2  {
3      list<double> l = {2.5, 3.5, 6.5, 7.5};
4
5      auto it1 = l.begin(); it1++; it1++;
6      l.insert(it1, 1.5); l.insert(it1, 10, -1); //what is l ?
7
8      vector<double> v = {1.1, 2.2, 3.3};
9      it1++; it1++; it1++;
10     l.insert(it1, v.begin(), v.end()); //what is v ?
11
12     it1 = l.begin();
13     for(int i = 0; i < 5; i++)
14         it1++;
15     l.insert(it1, v.begin(), v.end()); //what is l ?
16
17     it1 = l.begin();
18     auto it2 = l.end(); it2--; it2--;
19     for(int i = 0; i < 9; i++)
20         it1++;
21     l.erase(it1, it2); //what is l ?
22 }
```

Invalidation d'itérateurs

- Modifier le contenu d'un conteneur quand des itérateurs y sont instanciés peuvent les *invalid*er
- Le comportement de l'itérateur n'est pas celui attendu
 - Souvent, comportement indéterminé (déférencement, déplacement)
- Invalidations précisées dans la documentation
 - Avec `array`, pas d'invalidation possible

Exemple : `vector`

- Invalidés sur tous les éléments si augmentation de taille
 - Tous les éléments « suivants » si suppression
-
- En `Java`, lancement d'une exception

Accès aux éléments

- Performances varient fortement en fonction
 - du type de conteneur
 - de l'emplacement

Complexité

- `vector` et `array` : $\mathcal{O}(1)$ dans tous les cas
 - `list` : $\mathcal{O}(1)$ au début et à la fin, $\mathcal{O}(n)$ sinon
 - `deque` : $\mathcal{O}(1)$ dans tous les cas (moins bon que `vector`)
-
- Opérateur `[]` (pas de contrôle de borne) ou fonction `at` pour `array`, `vector` et `deque`
 - Pour `list`, on accède à un élément via un itérateur

Exemple

■ Fichier acces.cpp

```
1  int main()
2  {
3      array<int,5> a = {1, 1, 2, 3, 5};
4      vector<int> v = {1, 2, 3, 4, 5};
5      list<int> l = {5, 6, 7, 8};
6      deque<int> d = {10, 11, 12, 13, 14};
7
8      for(int i = 0; i < 4; i++)
9          cout << a[i] << " " << v[i] << " " << d[i] << endl;
10     cout << endl;
11
12     cout << l.front() << endl;
13     cout << l.back() << endl;
14
15     auto it = l.begin();
16     it++;
17     cout << *it << endl;
18     it++;
19     cout << *it << endl;
20 }
```

Augmentation de taille d'un `vector`

- Trois attributs particuliers dans `vector`

- 1 `size()` : nombre d'éléments stockés
- 2 `capacity()` : nombre d'éléments stockables sans augmentation de taille
- 3 `max_size()` : nombre maximum d'éléments stockables

- On a toujours $\text{size}() \leq \text{capacity}() \leq \text{max_size}()$

Ajout d'éléments

- Si $\text{size}() < \text{capacity}() : \mathcal{O}(1)$

- Sinon

- 1 Création d'un tableau de taille double ($\text{capacity}() \times 2$)
- 2 Recopie des éléments du « petit » tableau ($\mathcal{O}(n)$)
- 3 Destruction du petit tableau

- Même comportement pour `std::string`

■ Fichier vect-capac-increase.cpp

```
1  int main()
2  {
3      vector<int> v(3);
4      cout << "size_=" << v.size() << "_,_capacity_=" << v.capacity() << endl;
5      for(int i = 0; i < 20 ; i++)
6      {
7          v.push_back(2);
8          cout << "size_=" << v.size() << "_,_capacity_=" << v.capacity() << endl;
9      }
10     cout << endl;
11
12     v = vector<int>(3);
13     v.reserve(5);
14     cout << "size_=" << v.size() << "_,_capacity_=" << v.capacity() << endl << endl;
15
16     v = vector<int>(3);
17     v.push_back(2);
18     v.reserve(5);
19     cout << "size_=" << v.size() << "_,_capacity_=" << v.capacity() << endl << endl;
20
21     cout << v.max_size() << endl;
22 }
```

Adaptateurs

Généralités

- Conteneur aux propriétés d'ajout et d'accès particulières
- Implémentation non imposée par le standard

Opérations

- Ajout d'un élément (sans choisir « où »)
- Accès d'un élément (sans choisir « où »)
- Trois adaptateurs
 - 1 `stack` : pile (LIFO)
 - 2 `queue` : file (FIFO)
 - 3 `priority_queue` : file à priorité (<)

Adaptateur `stack`

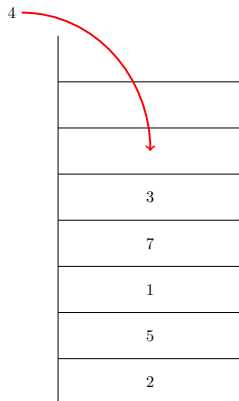
- Fournit une structure de donnée LIFO
- Construit par défaut ou sur base d'un conteneur séquentiel
 - Pour un usage exclusif, utilisez `list`
 - Efficace

Opérations

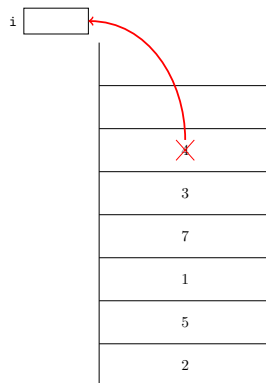
- `push` : met un élément au sommet de la pile
- `pop` : supprime l'élément au sommet de la pile
- `top` : accède au sommet de la pile

Illustration

```
stack.push(4);
```



```
int i = stack.top();  
stack.pop();
```



■ Fichier `stack.cpp`

```
1  int main()
2  {
3      stack<int> q;
4
5      for(int i = 0; i < 10; i++)
6          q.push(i * i);
7
8      while(! q.empty())
9      {
10         cout << q.top() << " ";
11         q.pop();
12     }
13     cout << endl;
14
15     for(int i = 0; i < 10; i++)
16         q.push(i * i);
17     q.top() = 99;
18
19     for(int i = 0; i < q.size(); i++) //sneaky loop is sneaky
20     {
21         cout << q.top() << " ";
22         q.pop();
23     }
24     cout << endl << q.size() << endl;
25 }
```

Adaptateur `queue`

- Fournit une structure de donnée FIFO
- Construit par défaut ou sur base d'un conteneur séquentiel
 - Pour un usage exclusif, utilisez `list`
 - Efficace

Opérations

- `push` : met un élément à la fin de la file
- `pop` : supprime l'élément au début de la file
- `front` : accède au début de la file
- `back` : accède à la fin de la file

■ Fichier `queue.cpp`

```
1  int main()
2  {
3      queue<int> q;
4
5      for(int i = 0; i < 10; i++)
6          q.push(i * i);
7
8      q.front() = 99;
9      q.back() = -99;
10
11     while(! q.empty())
12     {
13         cout << q.front() << " ";
14         q.pop();
15     }
16     cout << endl;
17 }
```


Adaptateur `priority_queue`

- Permet de maintenir un ordre sur les éléments
 - Opérateur `<` : « priorité »
 - Possibilité de fournir un comparateur par surcharge d'opérateur (Cf. Ch. 8)
- Construit par défaut ou sur base d'un conteneur séquentiel
 - Pour un usage exclusif, utilisez `deque`
 - Efficace

Opérations

- `push` : met un élément dans la file à priorité
- `pop` : supprime l'élément de plus haute priorité
- `top` : accède à l'élément de plus haute priorité
 - Ne peut pas être réaffecté

Exemple

■ Fichier `pqueue.cpp`

```
1  int main()
2  {
3      priority_queue<int> q;
4
5      for(int i = 0; i < 10; i++)
6      {
7          int r = rand() % 100 + 1;
8          q.push(r);
9          cout << "pushed_" << r << endl;
10     }
11     cout << endl;
12
13     while(! q.empty())
14     {
15         cout << q.top() << "_";
16         q.pop();
17     }
18
19     cout << endl;
20 }
```

Conteneurs associatifs

Généralités

- Structure de données associant une « clé » à une valeur.
- On accède aux valeurs via leur clé
 - Identique pour la suppression
- Souvent mis en œuvre à l'aide de tableaux associatifs et de tables de hachage
 - Tableau associatif : clés triées selon leur ordre « naturel »
 - Table de hachage : clés triées avec la fonction de hachage
- Les éléments sont parcourus en suivant l'ordre des clés

Exemple

- 1 Tableaux associatifs : `map`, `multimap`, `set`, `multiset`
- 2 Tables de hachage : `unordered_map`, etc.

Fonction de hachage : introduction

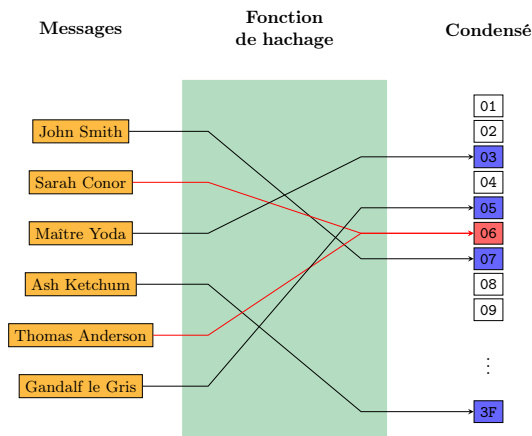
- Toute fonction associant un *message* (clé) à un *condensé*
 - Les messages ont une forme arbitraire (chaînes de caractères, bytes, etc.)
 - Souvent, les condensés sont naturels

Exemple

```
■ int i = f("John_smith"); //7
```

- Habituellement, « non inversibles »
- Si deux messages ont le même condensé : collision
- Exemples de fonctions de hachage : SHA-1, MD-5
- Domaine très important en informatique théorique
 - Question à 1M\$: « Do one-way functions exist ? »

Fonction de hachage : illustration



Le patron de classe `pair`

- Souvent, les « tuples » dans un conteneur associatifs sont stockés via `pair`
 - Spécialisation de `tuple`
 - Modélise une paire d'éléments de type donnés
- Dispose d'opérateurs
 - `==` et `!=` : comparaison de contenu
 - `<` et associés : comparaison lexicographique
- Patron utile lors de parcours itératif de conteneur associatif
 - « Pour chaque paire de `bro1, truc` » au sein de la `map` ...
- Fournit des conversions implicites
 - Très rare dans les templates
 - Souvent, utilisation de `make_pair`

Exemple

■ Fichier `pair.cpp`

```
1  int main()
2  {
3      pair<int, float> p1;
4      cout << "Init:_:" << p1.first << ",_" << p1.second << endl;
5
6      pair<int, double> p2(42, 0.123);
7      cout << "Initialized:_:" << p2.first << ",_" << p2.second << endl;
8
9      pair<char, int> p3(p2);
10     cout << "Implicitly_converted:_:" << p3.first << ",_" << p3.second << endl;
11
12     int n = 1;
13     int a[5] = {1, 2, 3, 4, 5};
14
15     // build a pair from two ints
16     auto p4 = make_pair(n, a[1]);
17     cout << "The_value_of_p3_is_" << "(" << p4.first << ",_" << p4.second << ")" << endl;
18
19     // build a pair from a reference to int and an array (decayed to pointer)
20     auto p5 = make_pair(ref(n), a);
21     n = 7;
22     cout << "The_value_of_p4_is_" << "(" << p5.first << ",_" << *(p5.second + 1) << ")"
23         << endl;
24 }
```


Le conteneur `map`

- Dictionnaire associant une clé à une valeur

Opérations

- Ajout / suppression : $\mathcal{O}(\log(n))$ (`insert` et `erase`)
- Accès : $\mathcal{O}(\log(n))$
- Les clés sont triées selon leur ordre naturel
 - Opérateur `<`
 - Possibilité de fournir un comparateur via surcharge d'opérateur (Cf. Ch. 8)

Exemple

■ Fichier map.cpp

```
1 void print(const map<char, int> &m)
2 {
3     cout << "Map_:_" << endl;
4     for(auto e : m)
5         cout << "(_" << e.first << "_,_" << e.second << "_) " << endl;
6     cout << endl;
7 }
8
9 int main()
10 {
11     map<char, int> m;
12     print(m);
13
14     m['S'] = 5; m['C'] = 10;
15     m['s'] = 2;
16     print(m);
17
18     m['S'] = m['D'];
19     print(m);
20 }
```

Accès aux éléments

- Accès via l'opérateur `[]`

Remarque importante

- Pas de contrôle de bornes
- Si `m[i]` n'existe pas, il est créé

- Fonction `at` : accès avec contrôle de bornes
 - Un élément non existant n'est pas créé (exception)
- Accès par itérateur
 - La fonction `find` retourne un itérateur sur un élément ayant une clé donnée, ou sur `end()` s'il n'existe pas
 - Défèrencement de `pair` via `*`
 - Obtention des clés / valeurs via les attributs `first` et `second`
 - Comportement de `*it = make_pair('S', 5);` non spécifié

Exemple

■ Fichier `map-access.cpp`

```
1  int main()
2  {
3      map<int, float> m;
4
5      for(int i = 0; i < 3; i++)
6          m[i] = i + 0.5;
7
8      cout << "m[5]= " << m[5] << endl;
9
10     for(auto e : m)
11         cout << "(" << e.first << ", " << e.second << ")" << endl;
12     cout << endl;
13
14     cout << (*m.find(2)).second << endl;
15
16     if(m.find(6) != m.end())
17         cout << (*m.find(6)).second << endl;
18     else
19         cout << "Key '6' does not exist" << endl;
20
21     for(auto e : m)
22         cout << "(" << e.first << ", " << e.second << ")" << endl;
23     cout << endl;
24 }
```

Autres conteneurs (1/2)

Conteneur `multimap`

- Conteneur à la signature identique à `map`, mais plusieurs valeurs peuvent être associées à une clé
- `find` fournit un itérateur sur l'un des éléments associés
 - Pas nécessairement le premier
- `equal_range` fournit tous les éléments associés à une clé
- `erase` efface tous les éléments associés à une clé (ou intervalle)

Conteneur `set`

- Conteneur identique à `map`, mais avec les clés uniquement
 - Idée : liste triée d'éléments uniques
- Les éléments sont constants

Autres conteneurs (2/2)

- `multiset` : signature identique à `set`, mais les doublons sont autorisés
 - Remarques identiques aux spécificités de `multimap`
- `unordered_map`, `unordered_multimap`, `unordered_set`, `unordered_multiset`
 - Même signature que les conteneurs associatifs associés, mais les clés sont spécifiées par une fonction de hachage
 - Possibilité de spécifier sa fonction de hachage via un foncteur

Remarque

- Créer une fonction de hachage ayant les propriétés désirées est parfois très difficile