

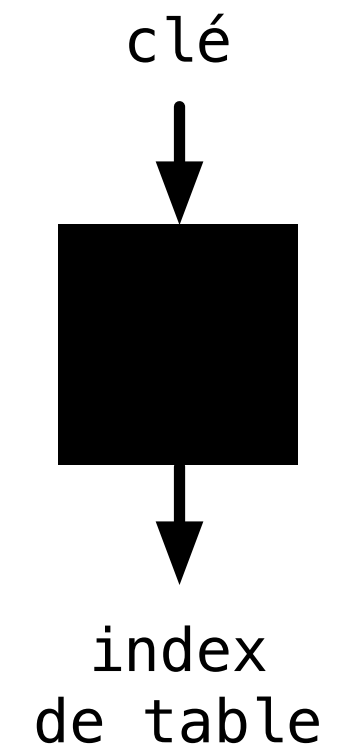
7.2. Fonctions de hachage



Fonction de hachage



- Une bonne fonction de hachage
 - est facile à calculer
 - minimise les collisions
 - distribue les adresses uniformément dans la table de hachage
 - utilise toute l'information fournie par la clé

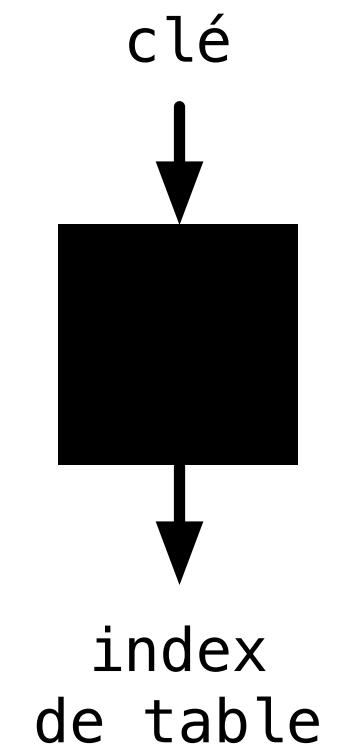


Par exemple, pour hacher des numéros de téléphone

👎 trois premiers chiffres : 021, 021, 021, 079, 079, 079, ...

079	123	45	67
021	601	12	34
021	789	01	23
079	456	78	90
079	987	65	43
021	345	67	89

Fonction de hachage



- Une bonne fonction de hachage
 - est facile à calculer
 - minimise les collisions
 - distribue les adresses uniformément dans la table de hachage
 - utilise toute l'information fournie par la clé

Par exemple, pour hacher des numéros de téléphone

	trois premiers chiffres : 021, 021, 021, 079, 079, 079, ...	079	123	45	67
		021	601	12	34
	trois derniers chiffres : 567, 234, 123, 890, 543, 789, ...	021	789	01	23
		079	456	78	90
		079	987	65	43
		021	345	67	89

Adressage par extraction



- Extraire des bits de la clé pour obtenir la valeur de hachage.
- Exemple : bits 3, 10, 18 et 23 (et un codage des bits par entier)
 $h(\text{"hello"}) = 110110\ 1\textcolor{brown}{0}0000\ \textcolor{brown}{1}10010\ 11\textcolor{brown}{0}010\ 110\textcolor{brown}{1}01_b = \textcolor{brown}{1011}_b = 11_d$



Facile à mettre en oeuvre



La valeur de hachage ne dépend pas de l'intégralité de la clé :

- $h(\text{"hello"}) = h(\text{"hello, world"}) = 11$

Une bonne fonction de hachage doit faire intervenir tous les bits de la clé.

Adressage par division



- $h(k) = k \text{ modulo } M$, avec M premier et éloigné des puissances de 2

👍 Bonne répartition

👎 Multiplie les collisions

👎 Dépend de M

- On la combinera avec d'autres méthodes.

Adressage par multiplication



- $h(k) = \lfloor M \cdot ((k \cdot A) \bmod 1) \rfloor$ avec A **réel** et tel que $0 \leq A < 1$.
- Le choix de A doit éviter les accumulations aux extrémités de la table. Knuth a montré que la valeur $A = \frac{\sqrt{5} - 1}{2}$ a de grande chance de bien marcher.

- Exemple avec $M = 10000$ et $A = \frac{\sqrt{5} - 1}{2} = 0,61803\dots$

$$\begin{aligned} h(123456) &= \lfloor 10000 \cdot ((123456 \cdot 0.61803) \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151 \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151 \rfloor \\ &= \lfloor 41.151 \rfloor = 41 \end{aligned}$$

Adressage MAD



- Adressage par **Multiplication, Addition et Division** (MAD) via une fonction $h(k) = (a \cdot k + b) \bmod M$ avec a et b **entiers** strictement positifs et a non multiple de M
- Exemple avec $M = 7$, $a = 8$, et $b = 5$
 $h(13) = (8 \cdot 13 + 5) \bmod 7 = 109 \bmod 7 = 4$
-  Avec a, b, M connus, vulnérable à une attaque par choix de clés donnant toutes le même hash, ce qui impacte les performances
- **Hachage universel** : choix aléatoire de la fonction de hachage indépendamment des clés pour annuler cet angle d'attaque.
 - $h(k) = ((a \cdot k + b) \bmod p) \bmod M$
 - En choisissant $p > k, \forall k \in \mathcal{K}$, et a et b dans $[1, p - 1]$, avec $a \neq b$.

Adressage par compression polynomiale



- On découpe la clé k en sous-chaînes de longueur 8, 16, 32 ou 64 bits, ce qui donne des coefficients $k_0, k_1, k_2, \dots, k_{n-1}$
- Pour une valeur de z fixe et non nulle, on calcule le polynôme $P(z) = k_0 + k_1 \cdot z + k_2 \cdot z^2 + \dots + k_{n-1} \cdot z^{n-1}$
- La fonction de hachage est dès lors
$$h(k) = (k_0 + k_1 \cdot z + k_2 \cdot z^2 + \dots + k_{n-1} \cdot z^{n-1}) \mod M$$
- Très bonne méthode pour le type string. Le choix de $z = 33$ donne au plus 6 collisions sur un ensemble de 50'000 mots anglais.

Hachage en C++



La bibliothèque standard C++ définit un foncteur générique de hachage `std::hash<KeyType>` qui retourne un résultat de type `size_t`. Il est spécialisé selon les types :

- Types **entiers** : cast de la valeur vers `size_t`
- Autres types **standards** (`float`, `double`, `string`, ...) : mises en oeuvre spécifiques
- Types **utilisateur** : à mettre en oeuvre par l'utilisateur en respectant ...
 - Toujours : Si $x == y$, alors $\text{hash}(x) == \text{hash}(y)$
 - Le plus souvent possible : Si $x \neq y$, alors $\text{hash}(x) \neq \text{hash}(y)$

class template

`std::hash` 

`<functional>`



```
template <class T> struct hash;
```

Default hash function object class

Unary function object class that defines the default hash function used by the standard library.

The functional call returns a hash value of its argument: A hash value is a value that depends solely on its argument, returning always the same value for the same argument (for a given execution of a program). The value returned shall have a small likelihood of being the same as the one returned for a different argument (with chances of collision approaching `1/numeric_limits<size_t>::max`).

Other function object types can be used as Hash for unordered containers provided they behave as defined above and they are at least *copy-constructible*, *destructible* function objects.

The default hash is a template class that is not defined for the general case. But all library implementations provide at least the following type-specific specializations:

header	types	header	types
<functional>	bool	<string>	string
	char		wstring
	signed char		u16string
	unsigned char		u32string
	char16_t	<memory>	unique_ptr
	char32_t		shared_ptr
	wchar_t	<vector>	vector<bool>
	short	<bitset>	bitset
	unsigned short	<system_error>	error_code
	int	<typeindex>	type_index
	unsigned int	<thread>	thread::id
	long		
	unsigned long		
	long long		
	unsigned long long		
	float		
	double		
	long double		
	T* (for any type T)		

Apart from being *callable* with an argument of the appropriate types, all objects of hash instantiations are *default-constructible*, *copy-constructible*, *copy-assignable*, *destructible* and *swappable*.

Users can provide custom specializations for this template with these same properties.

std::hash — Utilisation



```
#include <functional>
#include <iostream>
#include <string>
```

```
using namespace std;
```

```
int main() {
```

```
    // création du foncteur a l'avance
```

```
    std::hash<int> int_hash;
```

```
    cout << "hash(" << 1 << ") = " << int_hash(1) << endl;
```

```
    // objet foncteur temporaire
```

```
    cout << "hash(" << 'a' << ") = " << std::hash<char>>('a') << endl;
```

```
    // hachage de plusieurs strings
```

```
    std::hash<string> str_hash;
```

```
    for( string s : { "A"s, "AAAAA"s, "AAAAB"s } )
```

```
        cout << "hash(\"" << s << "\") = " << str_hash(s) << endl;
```

```
}
```

```
hash(1) = 1
hash(a) = 97
hash("A") = 3397809020744382953
hash("AAAAA") = 6458983483586025613
hash("AAAAB") = 3306170824323484213
```

Types définis par l'utilisateur



```
struct Transaction {
    std::string who;
    std::time_t when;
    long long amount;
};
```

Une classe définie par l'utilisateur

```
bool operator==(const Transaction &o) const {
    return who == o.who && when == o.when && amount == o.amount;
}
```

operator== est nécessaire pour distinguer les objets en cas de collision.

```
namespace std {
    template<> struct hash<Transaction> {
        std::size_t operator()(const Transaction& val) {
            using std::hash;
            size_t hashval = 17;
            hashval = 31*hashval + hash<std::string>()(val.who);
            hashval = 31*hashval + hash<std::time_t>()(val.when);
            hashval = 31*hashval + hash<long long>()(val.amount);
            return hashval;
        }
    };
}
```

Mise en oeuvre d'une fonction de hachage par spécialisation de std::hash

Commencer avec une constante non nulle

Faire contribuer tous les attributs pertinents pour operator==

Utiliser un nombre premier petit

Hachage modulaire



- `std::hash` retourne des valeurs de type `size_t` entre 0 et $2^{64}-1$.
- On veut des valeurs entre 0 et $M-1$ pour une table de hachage de taille M .
- On réduit la plage de valeurs modulo M .

```
size_t hash(Key key) {  
    return std::hash<Key>()(key) % M;  
}
```

7.3. Hachage dans la STL



Tables de hachage dans la STL



Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Structures
linéaires (ASD1)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Arbres (ASD1)
équilibrés (ASD2)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Tables de hachage



- TDA ensemble non trié
- Permet **insertion**, **suppression** et **recherche** en $O(1)$ (amorti)
- Plus rapide qu'un `std::set` ordonné qui donnait $O(\log(n))$

std::unordered_set

<unordered_set>

```
template < class Key,                                // unordered_set::key_type/value_type
          class Hash = hash<Key>,                    // unordered_set::hasher
          class Pred = equal_to<Key>,                // unordered_set::key_equal
          class Alloc = allocator<Key>               // unordered_set::allocator_type
        > class unordered_set;
```

Unordered Set

Unordered sets are containers that **store unique elements in no particular order**, and which allow for fast retrieval of individual elements based on their value.

In an `unordered_set`, the value of an element is at the same time its *key*, that identifies it uniquely. Keys are immutable, therefore, the elements in an `unordered_set` cannot be modified once in the container - they can be inserted and removed, though.

Internally, the elements in the `unordered_set` are not sorted in any particular order, but organized into *buckets* depending on their hash values **to allow for fast access** to individual elements directly by their *values* (with a constant average time complexity on average).

`unordered_set` containers **are faster than set containers** to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Iterators in the container are at least forward iterators.

std::unordered_map



std::unordered_map

<unordered_map>

```
template < class Key,                                // unordered_map::key_type
          class T,                                    // unordered_map::mapped_type
          class Hash = hash<Key>,                     // unordered_map::hasher
          class Pred = equal_to<Key>,                 // unordered_map::key_equal
          class Alloc = allocator< pair<const Key,T> > // unordered_map::allocator_type
        > class unordered_map;
```

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.

In an `unordered_map`, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped value* may differ.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their *key* or *mapped values*, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

`unordered_map` containers are faster than `map` containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the *mapped value* using its *key value* as argument.

Iterators in the container are at least forward iterators.

unordered_...



- **Performance** : unordered_set / unordered_map sont mises en oeuvre avec des tables de hachage et sont plus rapides pour l'insertion, suppression et recherche que leurs cousins set / map.
- **Non-ordonnées** : Il n'y a pas de notion d'ordre des clés.
 - Il n'y a pas de méthode pour accéder à la clé la plus petite ou la plus grande.
 - Les itérateurs parcourent les éléments du conteneur dans un ordre aléatoire.
- **Exigences pour les éléments** : Pour organiser les éléments les structures vont appliquer deux opérations aux éléments, qui doivent fournir des résultats valides:
 - Le class template hash appliqué à un élément doit fournir un code de hachage valide
 - operator== doit retourner true si deux éléments sont égaux
 - Par contre, pas besoin d'offrir operator< ni les autres opérateurs de comparaison