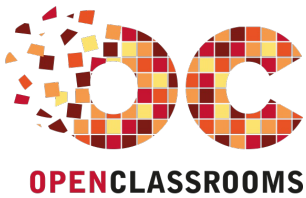
Les Conteneurs de la STL

Par Megaz



www.openclassrooms.com

Dernière mise à jour le 3/05/2011

Sommaire

Sommaire 2
Ce n'est pas parce que tu parles que tu es intelligent. 1
Les Conteneurs de la STL 3
La découverte par l'exemple 3
Un groupe de personnages 3
Appel des personnages 4
Le tableau associatif de la STL : std::map 5
Créer un std::map 5
Ajout et accès à une valeur associée d'une std::map 6
Utiliser une std::map en tant que conteneur 7
Une classe bien pratique pour manipuler des éléments contenus dans un conteneur STL : les itérateurs 9
Utilisation d'un itérateur de std::map 9
Quelques utilisations bien pratiques des itérateurs 10
Utilisation plus étendue des std::map 13
Un énorme terrain de Personnages... 13
Un troisième paramètre template bien caché 16
Il existe d'autres conteneurs associatifs de la STL! 19
Associer plusieurs valeurs associées à une clef d'un tableau associatif 19
Quand std::map nous donne trop d'informations... 20
Les autres conteneurs de la STL 21
Les adaptateurs de conteneurs de la STL 22
Les autres conteneurs séquentiels de la STL 25
Quelques fonctions pratiques que propose std::list 27
Une petite astuce de types ! 31
Q.C.M. 33
Itérateurs 33
Partager 38

Les Conteneurs de la STL

Par Megaz

Mise à jour : 03/05/2011
Difficulté : Intermédiaire Durée d'étude : 2 heures, 30 minutes

Il est nécessaire de connaître le contenu de la première partie et le début de la deuxième partie (notions de bases sur les classes) du cours de C++ de M@teo21/Nanoc. Ce tutoriel étant un (gros) complément au chapitre sur les tableaux dynamiques : `std::vector`.

Salut les zéros !

Vous avez découvert le conteneur `std::vector` sur le tutoriel de Nanoc et vous l'avez trouvé super pratique.
Mais comme dit dans le dernier chapitre du tutoriel C++ de M@teo21, il existe plein d'autres classes que le `std::vector` pour créer des tableaux !
Ces autres classes sont appellées "conteneurs" car elles ont toutes pour but de contenir des éléments. (Étonnant n'est-ce pas ?)

Comme indiqué sur ce schéma, en fonction de l'utilisation à laquelle votre tableau est utilisé, il existera peut-être une classe qui sera plus optimisée pour cette utilisation, ou qui vous fournira des fonctions plus appropriées.
C'est parti ! Laissez-moi vous présenter ces conteneurs !
Sommaire du tutoriel :

- La découverte par l'exemple
- Le tableau associatif de la STL : std::map
- Une classe bien pratique pour manipuler des éléments contenus dans un conteneur STL : les itérateurs
- Utilisation plus étendue des std::map
- Il existe d'autres conteneurs associatifs de la STL!
- Les autres conteneurs de la STL
- *Q.C.M.*

La découverte par l'exemple

Un groupe de personnages

Imaginez un jeu où l'utilisateur peut créer des personnages ayant chacun un nom.
Comment coder cet exemple avec un tableau classique ou un `std::vector` ?
Supposons que nous utilisons la même classe Personnage du tutoriel de Nanoc et M@teo21 (Moi je la trouve très sympatique cette classe). Au stade où les personnages ont un nom et sont capables de se présenter.

Code : C++

```
class Personnage
{
    public:
 Personnage(std::string nom);
        void recevoirDegats(int degats);
 void boirePotionDeVie(int pvAjoutes);
        void coupDePoing(Personnage& cible);

        std::string getNom() const;
```

```
        void sePresenter() const;

    protected:
        int m_vie;
        std::string m_nom;
};
void Personnage::sePresenter() const
{
    cout << "Bonjour, je m'appelle " << m_nom << "." << endl;
    cout << "J'ai encore " << m_vie << " points de vie." << endl;
}
```

L'utilisateur pourra créer des personnages par leur nom dans un `std::vector` :

Code : C++

```
int main(int argc, char ** argv)
{
 string nom;
 vector<Personnage> groupe;
 while(1)
 {
  cout << endl << "Entrez le nom du Personnage que vous voulez
creer" << endl;
  cin >> nom;
  groupe.push_back(nom);
  cout << "Voici maintenant votre groupe de personnages : " << endl;
  for(int i = 0;i<groupe.size();i++)
  {
   cout << "<Perso numero " << i << ">" << endl;
   groupe[i].sePresenter();
  }
 }
 system("PAUSE");
 return 0;
}
```

Il pourrait alors créer les personnages du nom qu'il veut, comme Bob, ou Michael_Jordan. 

Appel des personnages

Imaginons maintenant qu'après une longue série de combat, l'utilisateur veut savoir combien de vie il reste à Michael_Jordan. On peut créer une fonction affichant tous les personnages et il trouvera bien ce qu'il veut dedans. Mais évidemment ce n'est pas une bonne méthode si le groupe contient 500 personnages (j'envisage de grandes armées moi ).

Comme chaque Personnage a un numéro dans le tableau (un indice), l'utilisateur pourrait *retenir* le numéro de Michael_Jordan et nous créerions une fonction du style :

Code : C++

```
void presentation(const vector<Personnage> & groupe, int
numeroDuPerso)
{
 groupe[numeroDuPerso].sePresenter();
}
```

Ayant retenu que Michael Jordan avait le numéro 23 dans le groupe, on écrirait dans le main `presentation(groupe,23);`
Mais évidemment l'utilisateur ne pourra pas se rappeler des indices de tous les Personnages... Il ne connaît juste que leur nom.
Alors pourquoi ne pas surcharger la fonction `presentation` prenant comme paramètre un `std::string` ?

Le problème est là : à quel indice est placé Michael_Jordan ? Il faudra donc rechercher ce Personnage dans le groupe :

Code : C++

```
void presentation(const vector<Personnage> & groupe, string nom)
{
 for(int i = 0;i < groupe.size();i++)
 {
  if(groupe[i].getNom() == nom)
  {
   groupe[i].sePresenter();
   break;
  }
 }
 if(i == groupe.size())
  cout << "Il n'existe pas de Personnage avec ce nom" << endl;
}
```

Cette technique fonctionne bien entendu mais si le tableau a 500 personnages... La performance ne sera pas géniale : une boucle de 500 itérations maximum pour juste trouver un Personnage, ce n'est pas utile. Pour info, la complexité de cet algorithme est en O(n).

Voilà un exemple où le `std::vector` n'est pas adapté à la situation  , la STL fournit une classe de tableau associatif prévue pour des cas comme ça !
Attention voici ... `std::map` !

Le tableau associatif de la STL : std::map

Créer un std::map

Nous allons créer un autre tableau qui permettra de savoir à quel indice a été placé quel nom. Imaginez cela comme un tableau classique où les indices sont remplacés par des `std::string` :

Clef ⇒		Valeur associée
Bob	⇒	0
Georges	⇒	1
...	⇒	...
Michael_Jordan	⇒	23

Nous utiliserons alors ce tableau pour connaître l'indice auquel le personnage correspondant a été placé afin d'y accéder.
Je vais tout de suite vous montrer comment utiliser `std::map`.
Bien entendu il faudra inclure :

Code : C++

```
#include <map>
```

La création du tableau se fera comme ceci : 

Code : C++

```
map<string, int> indices;
```

Observons les paramètres template de `std::map` :

- La première classe à indiquer est appelée la classe de **clef** (en anglais **key value**), c'est la classe qui est utilisée pour accéder aux éléments. Pour chaque clef correspondra **une et une seule valeur associée**.
- La deuxième classe sera la classe de **valeur associée** (en anglais **mapped value**, d'où le nom). C'est la classe des éléments contenus dans le tableau.

En l'occurence ici nous utilisons une `std::string` pour obtenir un `int` qui est l'indice auquel le Personnage correspondant a été placé.

Ajout et accès à une valeur associée d'une std::map

L'avantage maintenant est que les éléments sont accessibles par leur nom. Ajouter un élément est très facile ! 
Voyez par vous-même.
En dessous du `push_back` j'écrirai cette ligne pour pouvoir accéder à l'élément créé par son nom :

Code : C++

```
indices[nom] = groupe.size() - 1;
```

(Après l'ajout, le Personnage a été placé à l'indice `size() - 1`)

Moi, quand j'ai vu ça, j'ai trouvé ça très logique comme écriture

Citation : Moi

> La valeur associée à la clef "nom" du tableau "indices" est égale à "groupe.size() -1"

ou encore :

Citation : Moi

> "indices" à "nom" = "groupe.size() - 1"

Cela s'utilise comme un tableau en C !

Code : C++

```
double tab[5]; //Création d'un tableau C
tab[3] = 25.1101992; //Ajout d'une valeur
```

Maintenant je peux utiliser ma `std::map` pour coder la surcharge de la fonction `presentation` que j'avais écrite :

Code : C++

```
void presentation(vector<Personnage> & tab, map<Personnage*> &
indices, string nom)
{
 int indice = indices[nom];
```

```
        tab[indice].sePresenter();
    }
```

Simple n'est-ce pas ?
L'avantage des conteneurs STL est qu'ils s'utilisent tous un peu de la même manière, on accède à un élément d'un `std::map` de la même manière qu'avec un `std::vector` : avec l'opérateur [].

Néanmoins, pour `std::map`, il existe une différence dans le comportement de cet opérateur, je vous en parlerai un peu plus tard après un exemple.

La complexité de l'opérateur [] est de $O(\log_2 n)$, contrairement à notre recherche dans un vector qui était de O(n). Pour des grands tableaux le changement sera très efficace !
Cependant l'accès à une valeur dans un tableau par un indice est le plus rapide possible : O(1). Si vous voulez utiliser un accès rapide aux éléments, utilisez une table de hachage. Celles-ci ont une complexité de O(1) en moyenne, même si le pire des cas est en O(n). Pour plus d'infos : lisez cette page sur wikipédia ou bien entendu les tutoriels de zéros traitant de ce sujet.

Vous pourriez très bien créer une autre `std::map` qui permettra par exemple d'appeler un Personnage par son numéro de téléphone !

Code : C++

```
map<int,int> annuaire; // Le premier int étant le numéro, le deuxième étant l'indice
annuaire[027336572] = 23; //Waw, le numéro de téléphone de Michael Jordan o_O en base 8 !
```

Utiliser une std::map en tant que conteneur

`std::map` est un conteneur me direz-vous, alors à quoi sert de garder encore un `std::vector` ?
Bien entendu, vous ne pourrez plus accéder aux éléments par un indice car justement on ne sait pas à quel indice notre élément recherché a été placé.
Alors, comment ferait-on ?

Been, comme ceci ? En ajoutant les éléments comme cela ?

Code : C++

```
std::map<string, Personnage> groupe;
groupe["Jean-Claude"] = Personnage("Jean-Claude");
```

Bien vu mais il faut bien se mettre en tête que ce code fera ceci :

1. Il créera un Personnage temporaire appelé Jean-Claude via le constructeur surchargé avec `std::string`.
2. Il essaiera d'accéder à l'élément "Jean-Claude" du groupe via l'opérateur []. Deux cas se présentent alors :
    - L'élément existe déjà, il renvoie une référence vers cet élément.
    - L'élément n'existe pas, **il en crée un en utilisant le constructeur par défaut** et renvoie une référence vers cet élément.

3. L'opérateur = est appelé, les données du personnage temporaire Jean-Claude sont copiées dans celles du Personnage du tableau qui est l'élément qui a été renvoyé par l'opérateur [].
4. Le personnage Jean-Claude temporaire sera détruit à la fin du bloc.

Deux Personnages seront alors créés alors qu'un seul aurait suffit.
(D'ailleurs si vous mettez quelques cout dans les constructeurs par défaut, de copie et avec paramètre(s) vous verrez qu'il crée beaucoup plus que deux Personnages...).
Cependant vous pouvez utiliser cette méthode si créer beaucoup de Personnages ne vous dérange pas.

Je trouve que cette manière de faire pour ce code convient très bien à la phrase que j'avais dite plus haut :

Citation

> la valeur associée à la clef "Jean-Claude" du tableau "groupe" est égale à "Personnage("Jean-Claude")"

Effet : peu importe s'il existait déjà une valeur associée pour cette clef, après cette instruction, la valeur associée sera "`Personnage("Jean-Claude")`". A nouveau, comme un tableau en C :

Code : C++

```
double tab[5]; //Création d'un tableau C
tab[3] = 25.1101992; //Ajout d'une valeur
tab[3] = 24.1101996; //Modification de la valeur
```

Comme vous pouvez le voir, l'opérateur [] **crée un objet** avec le constructeur par défaut si aucune valeur n'est associée à la clef entrée, ainsi écrire `groupe["Jean-Claude"];` suffit pour créer un élément.
Contrairement à l'opérateur [] de `std::vector` qui lui vous aurait simplement fait bugger le programme pour cause d'accès à une case de mémoire non autorisée (essaiez de faire vect[5] avec un vecteur long de 3 si vous ne voyez pas ce que je veux dire). Par contre la fonction membre `map::at` fonctionne exactement de la même manière qu'avec `std::vector`.

J'utiliserai alors un **`std::map<string, Personnage*>`** afin de contenir mes éléments, que je créerai dynamiquement, un Personnage de créé avec **`new`** et ensuite je garde un pointeur de celui-ci dans le tableau (Un peu à la Java ^^).
Voici alors comment j'ajouterai Bob à mon tableau de Personnage :

Code : C++

```
std::map<string, Personnage*> groupe;
groupe["Bob"] = new Personnage("Bob");
```

Ma fonction `presentation` devient alors toute simple :

Code : C++

```
void presentation(map<Personnage*> & groupe, string nom)
{
 groupe[nom]->sePresenter();
}
```

Mais maintenant, une question se pose très vite : comment faire pour accéder à tous les éléments du tableau ?
Je ne connais pas les noms de tous les Personnages !

Vous pourriez déjà le faire : en créant un tableau `std::vector<string>` contenant tous les noms possibles de la `std::map`.
Mais bien entendu, ce n'est pas la bonne manière de faire ! La STL a pensé à tout ! Comme d'habitude j'ai envie de dire.

Une classe bien pratique pour manipuler des éléments contenus dans un conteneur STL : les itérateurs

Voici la solution à notre problème !

heuu, mais, c'est quoi un i-té-ra-teur ?

C'est une classe qui a été créée spécialement pour parcourir, pour itérer les éléments d'une séquence, en l'occurence ici la séquence est `std::map`. Ice_Keese en parle dans son tutoriel.
Il vous dit qu'un itérateur est comme un pointeur vers l'élément de la séquence.
Et avec un pointeur on peut manipuler les éléments !
Bon assez parlé de ses mérites, voici comment créer un itérateur de la STL :

Code : C++

```
map<string, Personnage*>::iterator it;
```

Tous les conteneurs de la STL contiennent des fonctions renvoyant ou demandant un itérateur. Celles qui vont nous intéresser ici sont :

- `groupe.begin()` qui renvoie un itérateur vers le premier élément de la séquence
- `groupe.end()` qui renvoie un itérateur vers le ~~dernier élément de la séquence~~ l'élément SUIVANT le dernier élément de la séquence.

La boucle permettant d'accéder à tous les éléments de notre `std::map` sera celle-ci :

Code : C++

```
for(it = groupe.begin() ; it != groupe.end() ; ++it)
```

L'itérateur commencera donc à `begin()` et se terminera quand il sera arrivé à `end()`. C'est pourquoi `end()` renvoie un itérateur vers l'élément suivant au dernier élément de la séquence, pour arrêter la boucle quand il arrivera à cet itérateur. Si la boucle s'arrêtait au dernier élément, celui ne serait pas traité.

Remarquez que notre itérateur possède l'opérateur ++ surchargé, ce qui nous permet de facilement parcourir la séquence !

Bon maintenant qu'on sait parcourir la séquence, on aimerait bien savoir utiliser notre itérateur !

Utilisation d'un itérateur de std::map

Les itérateurs sont un peu comme des pointeurs vers les éléments de la séquence. Vous penseriez peut-être que les éléments de la séquence de `std::map<string, Personnage*>` sont des Personnages ? Et ben non ! Ce sont des **paires** "nom-personnage", et cette paire est en fait un objet de la classe `std::pair<string, Personnage*>` !

Un objet de classe `std::pair` nous met à disposition deux variables pour accéder à ce qu'elle contient :

- `paire.first` qui ici contiendra la clef de notre élément, en l'occurence une `std::string`.
- `paire.second` qui ici contiendra la valeur associée à notre élement, en l'occurence un `Personnage*`.

La boucle qui permettera la présentation générale des élements de notre `std::map groupe` se fera alors comme ceci :

Code : C++

```
for(it = groupe.begin();it != groupe.end(); ++it)
{
 cout << "clef " << it->first << " Présentation : " << endl;
 it->second->sePresenter();
}
```

L'itérateur étant comme un pointeur sur une paire, on écrit `it->first` pour accéder à la clef,en l'occurence une `std::string`.
De même, on écrit donc `it->second` pour accéder à la valeur associée. Cette valeur associée étant un pointeur vers Personnage on écrit
`it->second->sePresenter();` pour appeller sa fonction `sePresenter()`.

Je vous ai dit que tous les conteneurs de la STL avaient leur propre itérateur. `std::vector` en fait bien entendu partie. Ainsi vous pouvez parcourir tous les élements d'un `vector<Personnage*>` de cette façon :

Code : C++

```
vector<Personnage*> groupe;
//Admettons qu'on le remplisse ici
vector<Personnage*>::iterator it;
for(it = groupe.begin() ; it != groupe.end() ; ++it)
 (*it)->sePresenter();
```

(les éléments d'un `vector<Personnage*>` sont des pointeurs sur `Personnage`, donc `it` se comporte comme un pointeur sur pointeur de Personnage, ce qui fait qu'on écrit `(*it)` pour accéder au pointeur et ainsi `(*it)->sePresenter();` pour appeler la fonction membre !)

Les itérateurs de la STL s'utilisent donc tous de la même façon peu importe le conteneur.

Quelques utilisations bien pratiques des itérateurs

map::find

Voici son prototype pour notre classe :

Code : C++

```
map<string, Personnage*>::iterator find(const string & clef);
```

Cette fonction pourrait paraître inutile pour une `std::map` car pour trouver un élément dans une `std::map` il suffit d'utiliser l'opérateur [].
Mais le problème est que celui-ci, comme je vous l'ai discrètement dit plus haut, **crée un élément** s'il n'existe pas de valeur associée à la clef demandée. Or souvent, on voudrait faire une autre action que de créer un nouvel élément si la clef utilisée n'existe pas encore.
Et ben la fonction `map::find` renvoie l'itérateur `end()` dans ce cas (Qu'elle est gentille la fonction `map::find` !).

Ainsi on peut l'utiliser de cette manière :

Code : C++

```
it = groupe.find(nom);
if(it != groupe.end())
 it->second->sePresenter(); //Ou encore groupe[nom]->sePresenter();
else
 cout << "Il n'existe pas de Personnage avec ce nom" << endl;
```

map::insert

Cette fonction -comme c'est étonnant- permet d'insérer des éléments dans un conteneur STL.

Rappellez-vous que les `std::map` contiennent des `std::pair` !
Il faudra alors insérer des paires dans le tableau.

Vous pouvez créer une paire simplement avec son constructeur : `pair<string, Personnage*> laPaire("Bob", new Personnage("Bob"))`, néanmoins il existe une fonction qui allège parfois la création d'une paire : `std::make_pair` qui renvoie une paire avec les deux éléments passés en argument. Voyez l'exemple.

Voici comment on ferait :

Code : C++

```
map<string, Personnage*> groupe1;
map<string, Personnage*> groupe2;

pair<string, Personnage*> paireDeBob("Bob", new Personnage("Bob"));
groupe1.insert(paireDeBob);
groupe2.insert(pair<string, Personnage*>("Alphonse", new
Personnage("Alphonse")));
groupe2.insert(make_pair("Zozor", new Personnage("Zozor")));
//utilisation de make_pair
```

Notre tableau est prêt à être utilisé. 

Cette surcharge de la fonction renvoie une `std::pair<map<string, Personnage*>::iterator, bool>` où

- `paire.first` contient un itérateur vers l'élément ajouté ou l'élément déjà existant.
- `paire.second` est un booléen indiquant si l'élément a été ajouté (**`true`**) ou si il existait déjà une valeur associée à la clef demandée (**`false`**).

Si `paire.second == false`, le tableau n'a pas été modifié étant donné que dans une `std::map` **chaque clef associe une et une seule valeur associée.**

Une autre surcharge pratique de cette fonction est celle qui permet d'insérer des éléments d'une map vers une autre. Imaginons que le goupe2 rejoigne le groupe1 :

Code : C++

```
groupe1.insert(groupe2.begin(), groupe2.end()); //Insère à groupe1
```

```
les élément DE groupe2.begin() À groupe2.end() -non compris-
```

Remarquons qu'une copie de tous les éléments de groupe2 a été faite pour créer des nouveaux éléments identiques dans groupe1 via le constructeur de copie de éléments contenus, les conteneurs STL utilise d'ailleurs très souvent les constructeurs de copies lors des insertions, si ne voulez pas autant de copies ou si celles-ci posent un problème, utilisez l'allocation dynamique comme j'ai fait ici. Les copies seront seulement des copies de pointeurs qui ont une taille ridicule.

map::empty, map::size, map::clear et map::erase

Pas besoin de présenter les 3 premiers, vous les connaissez déjà via le tuto de Nanoc. 
Je vous avais dit que beaucoup de fonctions sont communes à tous les conteneurs STL.

Quant à `map::erase`, il permet -on ne s'y attendait vraiment pas- de supprimer un(des) élément(s) d'une `std::map` avec ou sans itérateurs.
(Pour `map::clear et map::erase`, n'oubliez pas quelques **`delete`** si les éléments de votre tableau ont été créés, comme moi, dynamiquement. Je ne l'écrirai pas ici car je considère que d'autres pointeurs pointent encore vers les Personnages que je supprime du tableau).

Un exemple ?

Code : C++

```
//Le groupe 2 avait rejoint le groupe 1.. ça veut dire qu'il n'y a
plus personne dans le groupe 2
groupe2.clear();

groupe1.erase(groupe1.find("Bob")); // erase par itérateur
cout << groupe1.size() << endl; // 2

groupe1.erase("Alphonse"); // erase par clef
cout << groupe1.size() << endl; // 1

groupe1.erase(groupe1.begin(), groupe1.end()); // erase DE
nombres.begin() À nombres.end() -non compris-. Dans ce cas-ci,
identique à clear()
cout << groupe1.empty() << endl; // true
```

La surcharge "erase par clef" renvoie le nombre d'éléments qui ont été supprimés.

Observez la méthode de "plages d'éléments bornés par deux itérateurs" dans la troisième surcharge. On efface les éléments DE `nombres.begin()` À `nombres.end()` **non compris**. Heureusement que celui-ci n'est pas compris car on ne peut pas accéder à l'élément pointé par `nombres.end()`, par sa définition. Cette méthode est utilisée pour beaucoup de fonctions. On l'utilise même dans notre boucle de présentation générale ! "Présenter tous les éléments DE begin() à end() -non compris-".

map::swap

Cette fonction permet d'échanger **le contenu de deux `std::map`.**

Code : C++

```
//Le groupe 1 devient le groupe 2 et inversément
groupe1.swap(groupe2);
```

STL Algorithms

STL Algorithms est un ensemble de fonctions de la bibliothèque standard très pratiques et très efficaces (comme on dit, tu ne battras jamais la STL).
Pourquoi je vous en parle ici ? Car une grande partie de ces algorithmes utilisent des itérateurs ! Vous pouvez découvrir la liste de ces fonctions ici.

Maintenant, il faudrait que je vous parle d'un élément important sur les `std::map`. (Et vous apprendre de nouveaux concepts !) Laissez-moi vous montrez un exemple ! (Car c'est toujours plus sympa de se dire qu'on va apprendre quelque chose en sachant à quoi ça pourrait servir).

Utilisation plus étendue des std::map

Un énorme terrain de Personnages...

Imaginez que vous avez un énoooorme terrain à cases (genre 500 x 500 ou plus) Et vous avez une cinquantaine de Personnage placé sur ce terrain. Il peut arriver qu'une bombe tombe du ciel sur une case qui enlèvera de la vie à toutes les cases adjacentes à la case sur laquelle elle a été lancée. Comment coderiez vous un tel jeu ?

... Réflexion ...

- Je fais une grande matrice 500 x 500 de pointeurs vers Personnages (`Personnage* terrain[500][500] = {0};`), ceux-ci seront à NULL si personne n'est dessus ?
- Ceci utilisera bien entendu beaucoup trop de mémoire, de plus ce n'est pas très "C++" comme pensée.
- Une std::map ? Puisque c'est le nom de la partie... Une `std::map<int,Personnage*>` plus précisément, où le int est un nombre contenant le numéro de la case, de 1 à 250 000 ?
- Ceci est une bonne idée, mais il y a moyen de penser encore plus "POO" !
- Je sais ! On crée une classe `Coordonnees` qui sera utilisée comme ceci :

Code : C++

```
std::map<Coordonnees, Personnage*> terrain;
```

- Voilà ce que je voulais entendre ! (Si vous y avez pensé tout de suite, je n'en serai que ravi !).

Alors c'est parti on écrit vite fait une telle classe :

Code : C++

```
class Coordonnees
{
 protected :
  int _x;
  int _y;
 public :
  Coordonnees(int x = 0,int y = 0);
  Coordonnees(const Coordonnees &);
  int getX() const;
  int getY() const;
  Coordonnees & operator=(const Coordonnees &);
  Coordonnees operator+(const Coordonnees &) const;
```

```
};

//Constructeurs
Coordonnees::Coordonnees(int x,int y) : _x(x), _y(y)
{}
Coordonnees::Coordonnees(const Coordonnees & autre) : _x(autre._x),
_y(autre._y)
{}

//Accesseurs
int Coordonnees::getX() const
{
 return _x;
}
int Coordonnees::getY() const
{
 return _y;
}

//Opérateurs
Coordonnees & Coordonnees::operator=(const Coordonnees & autre)
{
 _x = autre._x;
 _y = autre._y;
 return *this;
}
Coordonnees Coordonnees::operator+(const Coordonnees & autre) const
{
 return Coordonnees(_x + autre._x, _y + autre._y);
}
```

(Comme vous pouvez le voir j'aime commencer mes variables membres par un underscore pour les différencier des fonctions).

J'ai redéfini le constructeur de copie et l'`operator=` pour que vous voyez qu'il est important qu'une classe utilisée en clef dans une `std::map` aie ces fonctions car il les utilise !
J'ai également créé un `operator+` qui permet d'additionner des Coordonnees car ceci sera bien pratique pour notre utilisation.

Alors on crée vite fait une `std::map` pour tester notre nouveau concept ! 

Code : C++

```
int main()
{
 Coordonnees pos1(5,10);

 map<Coordonnees, Personnage*> terrain;

 terrain[pos1] = new Personnage("Bob",100); //Un gentil personnage à
la case 5,10
 terrain[Coordonnees(4,9)] = new Personnage("Megaz",25); //Un blessé
à la case 4,9

 lancerBombe(terrain, pos1); //On lance une bombe sur la case de
Bob !

 system("PAUSE");
 return 0;
}
```

Sans oublier notre fonction lancerBombe ^^. Allons allons un peu d'exercice !

Secret (cliquez pour afficher)

Code : C++

```
void lancerBombe(map<Coordonnees, Personnage*> & terrain, const
Coordonnees & centre)
{
 //Baam les Personnages présents sur les cases adjacentes à
centre se prennent 10 dégats ! Même en diagonale !
 map<Coordonnees, Personnage*>::iterator it;
 for(int Dx = -1;Dx <= 1; Dx++)
  for(int Dy = -1;Dy <= 1;Dy++)
  {
   it = terrain.find(centre + Coordonnees(Dx,Dy));
   if(it != terrain.end())
    it->second->recevoirDegats(10);
  }

}
```

Et on teste vite fait notre code ! 
...
Pour avoir une beeelle insulte du compilateur... Voici mon insulte de Visual C++ 2010 :

Code : Console

```
error C2784: 'bool std::operator <(const _Elem *,const std::basic_string<_Elem,_Tra
```

Car, scoop du jour, **les éléments d'une std::map sont triés !** Ils ne sont pas gardés par ordre d'insertion !
Ainsi sans que vous le sachiez, notre vielle `std::map<string, Personnage*>` gardait ses éléments par ordre alphabétique de clef ! Ainsi `groupe.begin()` vous renvoyait toujours un itérateur vers la paire ayant le nom le plus au début de l'alphabet.
Et quand nous faisions la boucle `for(it = groupe.begin();it != groupe.end();++it)`, les éléments étaient parcourus dans l'ordre alphabétique. 

Comment l'ordinateur arrivait-il à trier les éléments ? Car il écrivait `nom1 < nom2` afin de comparer les éléments entre eux. Et puisque `std::string` a un **`operator<`** surchargé, celui-ci arrivait à trier le tableau.

Voilà ce qui nous manque à notre Classe Coordonnees ! 

Code : C++

```
class Coordonnees
{
 protected :
  int _x;
  int _y;
 public :
  Coordonnees(int x = 0,int y = 0);
  Coordonnees(const Coordonnees &);
         int getX() const;
  int getY() const;
  Coordonnees & operator=(const Coordonnees &);
  Coordonnees operator+(const Coordonnees &);

bool operator<(const Coordonnees &) const;
};
bool Coordonnees::operator<(const Coordonnees & autre) const
{
```

```
    return (_x != autre._x) ? (_x < autre._x) : (_y < autre._y);
    }
    //Les autres définitions
```

J'ai écrit un opérateur qui triera les éléments en regardant d'abord leur _x et ensuite leur _y.
Quand l'ordinateur triera les éléments il placera `pos1` avant `pos2` si `pos1 < pos2`.

Veillez à avoir bien écrit les deux **`const`** dans la fonction de comparaison, en effet si vous les oubliez, le compilateur vous criera dessus... Il est très strict le compilateur...

Bien sûr n'importe qu'elle fonction aurait été bien, cela dépend de l'utilisation que vous voulez en faire.

Il existe néanmoins une règle à appliquer ! Car `std::map` n'utilise pas l'**`operator==`** pour savoir si une clef correspond à une autre ! il se dit que
`!(a < b) && !(b < a)` ⇒ `a == b`
(La flèche "⇒" signifie "implique")

(Pour les littéraires : Si a n'est pas plus petit que b et que b n'est pas plus petit que a alors a est égal à b).

Autre définition (Pour les matheux littéraires 😁) : si vous voulez utiliser une certaine fonction **`operator<`** pour votre tableau, cet **`operator<`** doit être une **relation irréflexive et transitive**, c'est aussi ce qu'on appelle une **relation d'ordre strict**.
Une relation irréflexive : `a < a == false`. Pour tout a.
Une relation transitive : `a < b && b < c` ⇒ `a < c`. Pour tout a, b, c.

Si vous utilisez une fonction **`operator<`** qui renvoie par exemple toujours **`true`**, en plus de ne pas avoir un tableau trié, aucune clef ne correspondra pas à une clef, même la bonne ! En effet si on place un Personnage en **(5,1)** et qu'on recherche **(5,1)**, il vous renverra qu'il n'y a pas de Personnage en **(5,1)** car **(5,1) < (5,1)** selon la fonction de comparaison !
Une fonction renvoyant toujours **`false`** par exemple fera que toute clef sera égale à n'importe quelle clef, par exemple **(5,1) et (6,2)** : **(5,2)** n'est ni plus petit que **(6,1)** et **(6,1)** n'est pas plus petit que **(5,1)** => clef égales ! Ce qui fera un tableau associatif de un élément...

Si vous trouvez deux éléments différents qui satisfont la relation, c'est que votre fonction n'est pas appropriée.

`std::map` n'utilise évidemment pas deux fois l'**`operator<`** mais cette règle est facile à utiliser pour vérifier si votre fonction de comparaison sera utilisable.

Notre code marche enfin !
Une petite boucle afin de présenter les Personnages, pour vérifier qu'ils ont effectivement perdu des points de vie :

Code : Console

```
Je m'appelle Megaz, et il me reste 15 points de vie
Je m'appelle Bob, et il me reste 90 points de vie
```

Vous pouvez maintenant utiliser `std::map` avec n'importe quelle classe en tant que clef (à sémantique de valeur généralement 😁).

Un troisième paramètre template bien caché

(Savoir écrire des template n'est pas nécessaire pour la compréhension de ce qui suit, ne fuyez pas devant ce nom s'il vous est inconnu !)

Saviez-vous qu'il existe un troisième paramètre template à la classe `std::map` ?
Il s'agit en fait de la **classe de Comparaison** utilisée pour trier les éléments. C'est une **classe foncteur** (voir tutoriel de Ice_Keese si vous avez des problèmes avec ce mot) : le compilateur en fait n'écrit pas `a < b` pour savoir si a doit être placé avant b mais il écrit `compare(a,b)`. Où `compare` est un **objet** de la classe foncteur. Et un objet d'une classe foncteur est appellé un ... foncteur. On doit pouvoir écrire `compare(a,b)` avec le foncteur utilisé dans le paramètre.

Quelques exemples seront plus parlants.

La classe foncteur peut être une structure ou une classe ou encore un pointeur sur fonction (voir tutoriel de Nanoc si ça vous intéresse, personnellement je trouve les classes plus faciles à utiliser).
Si vous choisissez la classe (ou structure), la condition pour que celle-ci soit une classe foncteur est qu'elle ait son **`operator()` surchargé** :

Code : C++

```
struct Comparer1
{
 bool operator()(const Coordonnees & a, const Coordonnees & b) const
 {
  return a.getX() != b.getX() ? a.getX() < b.getX() : a.getY() <
b.getY();
 }
};
class Comparer2
{
 public :
  bool operator()(const Coordonnees & a, const Coordonnees & b)
const
  {
   return a.getY() != b.getY() ? a.getY() < b.getY() : a.getX() <
b.getX();
  }
};
bool fonctionComparaison(const Coordonnees & a, const Coordonnees &
b)
{
 return a.getX() != a.getX() ? a.getX() > b.getX() : a.getY() >
b.getY();
}
```

On créera les terrains de cette façon, en indiquant le troisième paramètre template.

Code : C++

```
//Utilisation d'une structure
map<Coordonnees, Personnage*, Comparer1> terrain;
//utilisation d'une classe
map<Coordonnees, Personnage*, Comparer2> terrain2;
//Utilisation d'un pointeur sur fonction
bool(*pointeurSurFonction)(const Coordonnees &, const Coordonnees &)
= fonctionComparaison;
map<Coordonnees, Personnage*, (bool)(*)(const Coordonnees &, const
Coordonnees &)> terrain3(pointeurSurFonction); //Le pointeur est
indiqué en paramètre dans le constructeur.
```

Les valeurs seront alors triées selon le foncteur utilisé.

Mécanique interne

En interne il fait ça :
`Comparer1` est la classe foncteur, on l'a indiqué dans les paramètres template.

`std::map` se demande si `obj1` est plus petit que `obj2`.
Il crée alors un foncteur via la classe foncteur. Il le crée via le **constructeur par défaut**, c'est comme ça, c'est la classe qui a choisit ça (et c'est logique).

Code : C++

```
Comparer1 leFoncteurDeComparaison;
```

Ensuite il veut comparer `obj1` et `obj2` par exemple. Pour ça il sait que le foncteur a l'**`operator`**`()` surchargé. Il fait donc comme ça :

Code : C++

```
bool plusPetit = leFoncteurDeComparaison(obj1, ob2);
```

Un foncteur est donc bien un **OBJET** de la classe foncteur

Un peu de clarification

D'ailleurs, les paramètres commencent à être longs, je vous conseille d'utiliser des **`typedef`** pour alléger l'écriture :

Code : C++

```
typedef map<Coordonnees, Personnage*, Comparer1> Terrain;
typedef Terrain::iterator TerrainIt;
typedef map<Coordonnees, Personnage*, (bool)(*)(const Coordonnees &,
const Coordonnees &)> TerrainFct;
typedef TerrainFct::iterator TerrainFctIt;
```

Par défaut, ce troisième paramètre template est égal à `std::less<ClasseDeLaClef>` qui est une classe foncteur dont l'**`operator`**`()` renvoie **`true`** si `a < b`, c'est pour cela que nous avons dû définir une fonction **`operator`**`<` pour notre classe de clef quand le 3ème paramètre template était laissé par défaut.

Vous pouvez trouver ici la liste des classes foncteurs de la bibliothèque standard.

Ainsi vous pouvez utiliser des classes foncteurs déjà prédéfinies comme `std::greater`, attention à la règle qu'il ne faut pas enfreindre citée plus haut !
Un paramètre `greater_equal` ne fournira pas une bonne `std::map` car `!(a >= b) && !(a >= b)` n'implique pas que `a == b`.

Une surcharge de constructeur

Hum, la classe crée le foncteur via la classe des paramètres template. Et il le crée en utilsiant le constructeur par défaut, comme j'ai dit plus haut.
Imaginons qu'on veuille qu'il utilise un autre foncteur que celui par défaut.
Et bien, tous les conteneurs STL qui ont une classe foncteur en paramètre template ont un constructeur commun !
Il prend un paramètre : le foncteur à utilisier.
Exemple :

Code : C++

```
Comparer1 monPetitFoncteur;
map<Coordonnees, Personnage*> terrain(monPetitFoncteur);
```

C'est une autre manière d'indiquer le foncteur de comparaison à utiliser.

Il existe d'autres conteneurs associatifs de la STL!

Comme on vous l'avait déjà dit, il existe plus que deux conteneurs STL, notemment 3 autres conteneurs associatifs, maintenant que vous connaissez `std::map`, les apprendre se fera très vite !

Associer plusieurs valeurs associées à une clef d'un tableau associatif

Imaginons le cas précédent, mais dans le cas où plusieurs personnages peuvent se trouver sur la même case. Vous pouvez créer une `map<Coordonnees, vector<Personnage*> >` pour envisager ce cas, les éléments seront ajoutés via des `push_back` et enlevés avec une boucle traditionnelle en O(n), pour peu d'éléments, cela ne pose pas de problèmes.

Cependant il existe un conteneur STL fait pour ça : `std::multimap`.
Cette classe s'utilise exactement de la même manière que `std::map` à quelques changements logiques dûs à sa définition :

- `multimap::insert` ne renverra pas une paire "itérateur - booléen" puisqu'un élément sera toujours inséré, cette fonction renverra juste un itérateur vers l'élément ajouté.
- `multimap::find` renverra un itérateur vers *un et un seul* des éléments trouvés à la clef voulue. Bien entendu `multimap::end` sera renvoyé si aucune valeur n'est associée à la clef demandée. Pour accéder à toutes les valeurs associées à une clef, voir la fonction `equal_range` juste en dessous.
- Par conséquent, l'opérateur [] n'existe plus pour `std::multimap`, il faudra utiliser `find` et `insert` à la place pour bien indiquer ce qu'on veut faire.

Et voici deux fonctions qui pourront vous être utile pour la manipulation d'une `std::multimap` :

multimap::count

Cette fonction renvoie le nombre de valeurs associées à une clef.
Elle existait déjà avec `std::map` mais j'ai trouvé inutile d'en parler car la valeur renvoyée était soit 0 soit 1. Cependant cette valeur renvoyée peut être converti implicitement en booléen ce qui permet de facilement vérifier la présence d'une clef dans la (multi)map :

Code : C++

```
if(! groupe.count("Bob"))
 cout << "Il n'y a pas de Bob dans le groupe !" << endl;
```

multimap::equal_range

Voici son prototype :

Code : C++

```
pair<itérateur, itérateur> multimap::equal_range(Clef &);
```

Cette fonction renvoie une paire "Itérateur de début - Itérateur de fin" pour une certaine clef.
Il suffira alors de parcourir les éléments depuis l'itérateur de début renvoyé jusqu'à celui -non compris- de fin.
Par exemple, pour demander à tous les personnages sur la case (5,10) de se présentez gentillement à la console :

Code : C++

```
#include <map> //multimap est défini au même endroit que map

typedef multimap<Coordonnees, Personnage*> MTerrain;
typedef MTerrain::iterator MTerrainIt;
typedef pair<Coordonnees, Personnage*> paireM;
```

Code : C++

```
//Création de deux Coordonnees
Coordonnees pos1(5,10);
Coordonnees pos2(2,3);
//Création du terrain
MTerrain terrain;
//Insertion de Personnages sur le terrain
terrain.insert(paireM(pos1, new Personnage("Bob",100)));
terrain.insert(paireM(pos1, new Personnage("Megaz",25)));
terrain.insert(paireM(pos2, new Personnage("Alphonse",2)));
//Recherches tous les Personnages en pos1
pair<MTerrainIt, MTerrainIt> resultat = terrain.equal_range(pos1);
//Affichage du résultat
for(MTerrainIt it = resultat.first; it != resultat.second; ++it)
 it->second->sePresenter();
```

Dans le cas où aucun élément ne correspond à la clef, la fonction renvoie une paire d'itérateurs pointant vers le même élément. La boucle ne se fera pas car la condition de fin est tout de suite accomplie.

Quand std::map nous donne trop d'informations...

Un jour je suis tombé sur ce cas :
J'avais créé un conteneur contenant des Personnages alloués dynamiquement. Par exemple notre terrain de là tantôt.
J'avais besoin de créer "quelque chose" qui permettait d'effectuer certaines actions sur une liste de Personnages, par exemple, tant que ces éléments sont dans la liste, leur donner 10pvs par tour. J'avais créé donc cette liste avec un `std::vector`, ne connaissant que cela au début :

Code : C++

```
vector<Personnage*> armee; //Conteneur des personnages (créés
dynamiquement)
vector<Personnage*> fontaineMagique; //Pointeurs indiquant quels
Personnages obtiendront le bonus
```

A chaque tour je parcourais la fontaine pour donner 10pvs à chacun de ses membres.

Maintenant que ceci était créé, j'aimerais bien ajouter des personnages à cette fontaine, et en enlever en sachant juste le personnage... Ajouter des éléments se ferait avec `push_back` (avec un test pour voir si le personnage n'est pas déjà dans la liste) et les enlever en faisant une boucle qui enlève l'élément s'il le trouve dans la fontaine.
Une fois que j'ai connu les `std::map`, je les ai trouvées très utiles pour ce problème.

On ajoute des pointeurs avec `insert` pour que l'élément ne soit pas ajouté s'il est déjà dans la liste, et on supprime les éléments avec `erase`.
Mais que mettre comme valeur de classe associée ? Un `bool` car il ne prend qu'un octet en mémoire ? On n'a pas besoin d'avoir de valeur associée : le simple fait d'être dans la fontaine permettait d'avoir le bonus de 10 pvs.

C'est alors que j'ai vu qu'il existait `std::set` qui est un conteneur STL où les éléments sont accédés par clef et où la valeur associée est... la clef !

Ainsi voilà comment j'ai finalement fait :

Code : C++

```
#include <set>
```

Code : C++

```
vector<Personnage*> armee;
set<Personnage*> fontaineMagique; //Utilisation d'un std::set, plus approprié
```

Code : C++

```
void ajouter(set<Personnage*> & fontaine, Personnage* perso)
{
 fontaine.insert(perso);
}
void enlever(set<Personnage*> & fontaine, Personnage* perso)
{
 fontaine.erase(perso);
}
```

La fonction à appeler à chaque tour serait celle-ci :

Code : C++

```
void donnerVie(set<Personnage*> & fontaine)
{
 for(set<Personnage*>::iterator it = fontaine.begin(); it != fontaine.end; ++it)
   (*it)->boirePotionDeVie(10);
}
```

(Rappellez-vous que l'itérateur "it" se comporte comme un pointeur sur un élément du conteneur qui ici est un pointeur sur personnage. On écrit donc `(*it)` pour accéder au pointeur et `(*it)->boirePotionDeVie(10);` pour appeler la fonction membre)

Il m'est déjà arrivé plein de fois où j'avais besoin de faire de telles actions.

Tout comme `std::map`, `std::set` possède un paramètre template de Comparaison pour trier ses éléments qui est à `std::less` par défaut. Ici ce sera le deuxième paramètre template.

Comme pour `std::multimap`, il existe une classe `std::multiset` qui fonctionne exactement de la même manière.
Alors, ne vous avais-je pas dit que tous les conteneurs s'utilisaient à peu près de la même façon ?

Les autres conteneurs de la STL

Et si je vous disais que, maintenant que vous connaissez tout sur `std::map` et surtout sur les itérateurs des conteneurs STL, apprendre les autres ne sera qu'une formalité ?
Les conteneurs STL sont séparés en trois parties :

- Les autres conteneurs *séquentiels* qui ont une interface similaire à `std::vector`, mais représentent néanmoins des

concepts différents et ne sont pas prévus pour le même usage. Il s'agit de `std::deque` et `std::list`.
- Les *adaptateurs* de conteneurs réduisant les possibilités d'un `std::vector` à quelques fonctions, vous pourriez très bien faire ces classes vous même, héritant de `std::vector` par exemple.
  Il s'agit de `std::stack`, `std::queue` et `std::priority_queue`.
- Et bien entendu les conteneurs associatifs ! Vous les connaissez bien maintenant.

Sachez qu'il existe encore un conteneur associatif dont je ne parlerai pas : `std::bitset`, il permet de stocker des bits sur juste l'espace suffisant : 8 bits sur un octet.

Et bien entendu, vous devrez inclure les headers pour chaque classe utilisée. 

Code : C++

```
#include <stack>
#include <deque>
#include <queue> // Qui contiendra aussi priority_queue :)
//etc.
```

Les *adaptateurs* de conteneurs de la STL

Ces classes sont faites pour n'être utilisées qu'avec quelques fonctions pour une utilisation bien précice :

std::stack (ou pile pour les anglophobes)

Cette classe est utilisée quand le tableau est réduit à une pile, par exemple une pile de feuilles (qui à la fin des examens s'est dispersée dans toute ta chambre).
Que peut-on faire avec une pile (à part la disperser...) ?

- Ajouter des éléments à son sommet. La fonction à utiliser pour cette action sera ... `stack::push`, elle se comporte exactement comme `push_back`...
- Accéder à son élément à son sommet. La fonction sera ... `stack::top`, elle se comporte comme `back` ou `front`. Mais ici on n'a pas l'embarras du choix car le seul élément auquel on peut accéder est l'élément au sommet (*at the top*).
- Supprimer l'élément au sommet, ce qui permettera d'accéder aux éléments en dessous. La fonction sera...`stack::pop`... Sans commentaires, cette fonction ne prend aucun paramètre et ne renvoie rien, elle ne fait juste qu'enlever l'élément *at the top*. Et si la pile est vide ben.. Elle reste vide...

Cette classe a bien entendu les fonctions habituelles comme `size` ou `empty`.

Petite subtilité, cette classe a quand même un paramètre template caché !
C'est la classe qui devra être utilisée pour contenir les éléments, un des trois conteneurs séquentiels. Par défaut, ce paramètre est `std::deque<T>` (où T est le type des éléments contenus).

std::queue (prononcez Quiouu) (ou file pour les francophones n'aimant pas dire "Je fais la queue")

Dans le même état d'esprit de simplification, voici la classe représentant une file : `std::queue`.
C'est comme une pile sauf qu'on ajoute les éléments à la fin de la file histoire que le premier arrivé soit le premier servi !
Ben oui, les éléments qui arrivent plus tard, ils n'ont qu'à faire la queue !

Ainsi, la fonction `stack::top` est remplacée ici par `queue::front` (Premier élément de la file, celui-ci qui va être traité) et `queue::back` (Dernier élément de la file, il attendra son tour !). La fonction `queue::push` ajoute un élément à la fin de la file

et `queue::pop` supprime le premier élément (après lui avoir donné son paquet de timbres bien entendu !).
Vous pouvez aussi choisir le type de conteneur utilisé bien que le paramètre par défaut est le plus efficace.

Si vous désirez un tutoriel beaucoup plus détaillé sur ces deux conteneurs ainsi que des exemples d'utilisation, regardez le tutoriel de Xavinou.

std::priority_queue (prononcez Pwraïowrity quiouu)

Cette classe est un peu plus développée que les deux autres que je viens de ~~présenter~~ platement décrire.
Vous pouvez ajouter des éléments avec `priority_queue::push` mais celui-ci ne vas pas toujours se placer au début comme une pile ou à la fin comme une file mais *à sa place*, de telle sorte que les plus importants éléments soient au début de la liste. Ainsi cela vous permet de toujours traiter en premier l'élément avec la priorité la plus grande (avec `priority_queue::top`).

Mais, comment cette classe sait quel élément a *de l'importance* ?

Il utilise la même technique que `std::map` pour trier ses éléments.
Alors, vous vous en souvenez ? Il utilise une classe foncteur de *Comparaison* qui permettra de savoir quel est l'élément le plus important entre deux. Cette classe foncteur était écrite dans le troisième paramètre template de `std::map`, et ben pour `priority_queue`, c'est aussi le 3ème paramètre template, le deuxième étant, rappelez vous, la classe utilisée pour contenir les éléments.

Je trouve que cette classe mérite un petit exemple, sous forme d'histoire (J'ai été inspiré là).

Code : C++

```
struct PvMin
{
 bool operator()(const Personnage & p1, const Personnage & p2) const
 {
  return p1.vie() > p2.vie();
 }
};
#include <queue>
int main()
{
 cout << "Il était une fois un petit village de Personnage
(symbolisé ici par un vector étant une priority_queue)" << endl;
 priority_queue<Personnage, vector<Personnage>, PvMin> village;
```

Voilà comment utiliser le paramètre template. Vous voulez vraiment la suite du code ? Ma petite histoire ?

Secret (cliquez pour afficher)

Code : C++ - La petite histoire

```
 cout << "Il contient un certain nombre d'habitants" << endl;

  village.push(Personnage("Bob")); //Bob, toujours en pleine santé
  village.push(Personnage("Viellard du coin",10)); // Le viellard
 du coin
  village.push(Personnage("Marie",75));
  village.push(Personnage("Sophie",75));//Les deux jumelles, aussi
 en forme l'une que l'autre
  village.push(Personnage("Gaston",60));//Le brave gars :)
```

```
 village.push(Personnage("Alphonse",90));//Le sportif :)

 cout << "Mais malheureusement les conditions de vie dans ce
village sont dures : chaque jour le personnage avec le moins de
santé s'exile tellement dures elles sont, observez par vous même
:" << endl;

 int i = 1;
 while(i<=3)
 {
  cout << "---" << endl;
  cout << "Jour " << i << endl;
  cout << "Aujourd'hui, on entend quelqu'un se présenter :" <<
endl;

  village.top().sePresenter();

  cout << "Il s'avère que cette personne a voulu quitter le
village ..." << endl;

  village.pop();

  cout << "Il reste " << village.size() << " habitants" << endl;

  i++;
 }
 cout << "Un nouvel habitant arrive dans ce village : " << endl;

 Personnage nouveau("Alfred",80);

 cout << "Poli, il se présente : " << endl;

 nouveau.sePresenter();

 village.push(nouveau);

 while(! village.empty())
 {
  cout << "---" << endl;
  cout << "Jour " << i << endl;
  cout << "Aujourd'hui, on entend quelqu'un se présenter :" <<
endl;

  village.top().sePresenter();

  cout << "Il s'avère que cette personne a voulu quitter le
village ..." << endl;

  village.pop();

  cout << "Il reste " << village.size() << " habitants" << endl;

  i++;
 }
 cout << "Le village est maintenant abandonné.. Il devient une
ville fantôme.." << endl;

 return 0;
}
```

Exécutez ce code après avoir rajouté une fonction `int vie()` aux Personnages, la console vous racontera une petite histoire.


Le test de comparaison d'importance ne se fait que lors de **l'ajout** d'un élément dans la `priority_queue`. Si un élément dans la file acquiert plus (ou moins) d'importance, il ne sera plus à sa place et la file ne fera plus ce qu'elle devra ! Par exemple, dans l'algorithme de Dijkstra, un élément dans la liste ouverte peut voir son poids descendre et donc sa

place devrait être changée dans la file. Si de tels cas apparaissent, il faut utiliser uen autre classe pour faire la file de priorité.
Exemple de problème :

Code : C++

```
struct Comp
{
 bool operator()(int * un, int * deux)
 {
  cout << "Comparaison" << endl;
  return *un < *deux;
 }
};
int main()
{
 int a(2), b(7);
 priority_queue<int*, vector<int*>, Comp> f;

 f.push(&a);
 cout << *f.top() << endl; //2 ... Normal
 f.push(&b);
 cout << *f.top() << endl; //7 ... Normal car l'élément a une plus grande importance
 b = 0;
 cout << *f.top() << endl; //0 ... Pas normal, cela devrait être 2

 return 0;
}
```

Cela devrait être l'adresse de a en premier dans la file. Mais quand on met la valeur de b à zéro, le conteneur ne se doute pas qu'une importance a changé dans ses éléments.
Pour éviter ce problème, il faut se dire que quand on ajoute un élément, le conteneur fait une comparaison avec les éléments existants pour placer le nouvel arrivant à sa place. Et c'est le seul moment où il fait des comparaisons.
Si on a besoin de faire changer les importances des éléments dans la file, il faut faire sa propre file de priorité (implémenté généralement sous forme de tas).

Les autres conteneurs *séquentiels* de la STL

std::deque (Double Ended Queue si vous voulez tout savoir ...)

`std::deque` est une classe qui s'utilise de la même façon que vector ! Les fonctions que vous connaissez pour vector pourront aussi être utilisées ici.
Néanmoins `std::deque` possède deux différences :

1. Un `std::vector` est optimisé pour que les nouveaux éléments soient ajoutés à la fin, c'est pourquoi il possède une fonction `push_back` et `pop_back`. `std::deque` est lui par contre optimisé pour que les nouveaux éléments soient ajoutés à la fin .. ou au début ! Il possède donc les fonctions `push_front` et `pop_front` en plus que `std::vector`.
2. Les éléments de `std::vector` sont stockés comme pour un tableau en c, **à des places continues de la mémoire** alors qu'avec `std::deque`, pas nécessairement. Cela n'a pas d'importance tant qu'on ne doit pas faire de conversions du style `std::vector` => tableau C-like. 

D'ailleurs, je vous informe que les itérateurs de `std::vector` et `std::deque` sont arithmétiques (*RandomAccess iterator*) !

Euh mais c'est quoi un itérateur arithmétique ?

Cela veut simplement dire que vous pouvez écrire ceci :

Code : C++

```
vector<int> v(10,2);
vector<int>::iterator it = v.begin();
it = it + 5;
it -= 5;
```

Utiliser des opérateurs arithmétiques pour se déplacer plus vite dans le tableau.
Ceux des tableaux associatifs par exemple avaient les opérateurs !=, ==, ++ (etc.) surchargés. Ceux-ci en a plus, c'est tout. Si vous voulez savoir quels itérateurs possèdent quelles surcharges d'opérateur, jetez un coup d'oeil à cette page. Vous pouvez même voir que ces *RandomAccess iterator* ont par exemple les opérateurs < et > surchargés.
Les itérateurs de `std::list` et des tableaux associatifs étaient appelés *Bidirectional iterator* car avec l'un de ces itérateurs on peut accéder à l'itérateur suivant et précédent uniquement (avec les opérateurs ++ et --).

La fonction `std::advance(itérateur, nb)` peut-être utilisée pour avancer un itérateur de la valeur nb. Si l'itérateur est arithmétique, C++ utilisera l'opérateur += ou -= sinon il fera une boucle de ++ ou --.

Bref, utilisez `std::deque` si vous avez besoin d'insérer des éléments autant au début qu'à la fin du tableau.

Et si vous avez besoin d'insérer des éléments au milieu ... Voici le dernier conteneur de la STL :

std::list

Un élément stocké dans une `std::list` prendra plus de place en mémoire qu'un `std::vector` mais vous fournira des performances bien meilleures si vous avez besoin d'effectuer beaucoup de déplacements à l'intérieur de la liste : élimination, déplacement de bloc d'une liste à l'autre ou dans la même, ajout d'éléments après un certain autre etc.

Les fonctions principales de `std::list` sont `list::insert` et `list::erase` que vous connaissez bien.
Cependant `list::insert` ne s'utilise pas exactement de la même manière, voici ses prototypes :

Code : C++

```
itérateur insert(itérateur position, const Valeur & x);
void insert(itérateur position, size_t nombre, const Valeur & x);
void insert(itérateur position, itérateur debut, itérateur fin);
```

`position` est un itérateur qui indique **avant** quel élément la valeur doit être insérée.
Pour la deuxième surcharge, cela permet d'indiquer combien d'éléments identiques ajouter et pour la troisième cela permet d'ajouter les éléments avant `position` DE `debut` À `fin` -non compris-, de la même manière qu'on avait fait avec `std::map` :

Code : C++

```
//si groupe1 et groupe deux sont des std::map
groupe1.insert(groupe2.begin(), groupe2.end()); //On ajoute le
groupe 2 au groupe 1
//si groupe1 et groupe2 sont des std::list
groupe1.insert(groupe1.end(), groupe2.begin(), groupe2.end()); //On
ajoute le groupe 2 à la fin du groupe 1
```

Un peu de généricité

Vous savez quoi ? `std::vector` (et `std::deque` bien sûr) possède **exactement** la même fonction ! Mais je vous rappelle que `std::vector` est optimisé pour des insertions à la fin seulement (et `std::deque` pour des insertions à la fin ou au début).
Une utilisation pratique est la fusion de deux tableaux comme montré ci-dessus ou encore cette utilisation ci-dessous que je trouve très pratique : 

Code : C++

```
int tabC[5] = {5, 2, -99, 12, 0xFF9900}; //Un tableau C-like
initialisé avec une liste d'initialisation !
vector<int> tab;
tab.insert(tab.end(), tabC, tabC + 5); //Waw ! Le tableau C-like
est considéré comme un itérateur ! On insère avant end() DE tabC à
tabC + 5 -non compris- ! Que c'est pratique !
```

Et encore pour avoir un code plus clair. Il existe un constructeur surchargé pour les itérateurs qui prend une paire d'itérateur pour savoir avec quoi le conteneur doit être rempli. Le code ci-dessus peut s'écrire en deux lignes :

Code : C++

```
int tabC[5] = {5, 2, -99, 12, 0xFF9900}; //Un tableau C-like
initialisé avec une liste d'initialisation !
vector<int> tab(tabC, tabC + 5); //Initialisé avec les itérateurs !
```

Cette surcharge qui permet un raccourci de la fonction `insert` est présente dans **tous les conteneurs**  (même les associatifs !)

 Mais, si je voulais utiliser le constructeur qui prenait comme paramètre un foncteur ? Et aussi celui-là ?

Il existe encore un constructeur qui mixe les deux. D'abord les itérateurs, et puis les foncteurs (après tous, vous les avez appris dans cet ordre la non ?) :

Code : C++

```
map<Coordonnees, Personnage*> terrain(m1.begin(), m1.end(),
monFoncteur);
```

Quelques fonctions pratiques que propose std::list

list::assign et list::resize

Ce sont **exactement** les mêmes fonctions que celles de `std::vector`. Sachez néanmoins que `assign` peut être utilisée avec des itérateurs (maintenant que vous les connaissez) :

Code : C++

```
list<string> maListe;
list<string> maListe2;
maListe.assign(5,"Blaaaaaaaaa"); //La première liste contient 5
Blaaaaaaaaa maintenant.
maListe2.assign(maListe.begin(), maListe.end()); //Une copie de
```

```
maListe
```

list::splice

Cette fonction est une manière plus rapide de déplacer-effacer des éléments à traver une même liste ou plusieurs. Vous pouvez l'utiliser de 3 manières différentes, voyez par vous-même (exemple platement pris de mon site de référence) :

Code : C++

```
list<int> liste1;
list<int> liste2;

for(int i = 1;i<5;i++)
 liste1.push_back(i); //liste1 : 1,2,3,4
for(int i = 20;i<25;i++)
 liste2.push_back(i); // liste2 : 20,21,22,23,24

liste<int>::iterator it = liste1.begin(); //it pointe vers 1

it++; //it pointe vers 2

liste1.splice(it, liste2); // La liste 2 sera placée toute entière
dans la liste1 juste avant l'élément pointé par it;
//liste1 : 1,20,21,22,23,24,2,3,4
//it pointe toujours vers 2

cout << liste2.empty(); // true, la liste a été été vidée

liste2.splice(liste2.begin(), liste1, it);//Place avant
liste2.begin() l'élément de liste1 pointé par it
//liste1 : 1,20,21,22,23,24,3,4
//liste2 : 2

it = liste1.begin(); // it pointe vers 1
advance(it,5); //it pointe sur 24

liste1.splice(liste1.begin(), liste1, it, liste1.end()); //Prendre
les éléments de liste1 DE it À liste1.end()-non compris- et les
placer avant liste1.begin()
//liste1 : 24,3,4,1,20,21,22,23

it = liste1.begin();//Pointe 24
advance(it,1);//pointe 3

liste<int>::iterator it2 = it; // Pointe 3
advance(it,3); //pointe 20

liste2.splice(liste2.end(), liste1, it, it2); //prendre les éléments
de liste1 DE it À it2-non compris- et les placer avant liste2.end()
//liste1 : 24,20,21,22,23
//liste2 : 2,3,4,1
```

list::remove

Permet de supprimer tous les éléments d'une liste ayant une certaine valeur. Exemple : `liste.remove(5);` supprimera tous les éléments égaux à 5 dans la liste. Cette fonction utilise l'**`operator`**`==`.

list::remove_if

Permet de supprimer tous les éléments d'une liste satisfaisant une certaine condition. Le paramètre à indiquer est un ***prédicat*** prenant comme paramètre un élément de la liste.

Un prédicat est simplement un foncteur qui renvoie un booléen ou quelque chose de convertible en un booléen.
Si vous vous embrouillez avec les noms "foncteurs" et "prédicats", voici un petit résumé :

- Un foncteur (appelons le "f") est un objet avec lequel on peut écrire **`f(quelqueChoseOuRienDuTout)`**. Par exemple f(5), f(a,b) ou encore f(). Les fonctions sont donc des foncteurs.
- Une classe ayant l'**`operator()`** surchargé créera des instances qui sont des foncteurs (par la définition d'un foncteur). Cette classe sera appelée "classe foncteur".
- Un prédicat est un foncteur renvoyant un booléen ou quelque chose de transformable en booléen. On pourra alors qualifier de "classe prédicat" des classes créant des prédicats.

Remarquez que parfois le mot "foncteur" est utilisé pour "classe foncteur" (mais dans ce cas on ne peut pas différencier la classe et les objets ...). Idem pour "prédicat".

Par exemple pour enlever tous les Personnages avec une vie de plus de 50 d'une liste de Personnage :

Code : C++

```
bool vie_plus_grande_que_50(const Personnage & p)
{
 return p.vie() > 50;
}
class Classe_vie_plus_grande_que_50
{
 public:
  bool operator()(const Personnage & p)
  {
   return p.vie() > 50;
  }
};
```

Code : C++

```
list<Personnage> liste;
//Remplissage avec push_back, push_front, insert, assign etc.
liste.remove_if(vie_plus_grande_que_50); //en utilisant une fonction
comme prédicat :

Classe_vie_plus_grand_que_50 leFoncteur; //On crée un foncteur, je
dirais même un prédicat
liste.remove_if(leFoncteur); //en utilisant un objet d'une classe
comme prédicat, la fonction appellera son operator()
```

Vous voyez l'intérêt du foncteur, la fonction pourra écrire leFoncteur(a,b) pour savoir si les deux éléments son égaux, car, puisque c'est un foncteur, il a son **`operator()`** surchargé.
Et vu que son **`operator()`** renvoie un booléen, on peut même le qualifier de prédicat !

L'utilisation de la classe permet beaucoup de choses ! Comme par exemple ceci :

Code : C++

```
class Classe_vie_plus_grande
{
 private :
```

```
  int _val;
 public:
  Classe_vie_plus_grande(int val) : _val(val)
  {}
  bool operator()(const Personnage & p)
  {
   return p.vie() > _val;
  }
};
```

Code : C++

```
Classe_vie_plus_grande monFoncteur(50);
liste.remove_if(monFoncteur);
```

Ou en une ligne :

Code : C++

```
liste.remove_if(Classe_vie_plus_grande(50));
```

list::unique

Permet de supprimer tous les doublons consécutifs d'une liste.
C'est à dire qu'après avoir exécuté cette fonction, la liste ne contiendra plus deux éléments égaux l'un à côté de l'autre.

- Si vous n'indiquez aucun paramètre, l'**operator**== sera utilisé.
- Si vous voulez comparer des éléments selon un autre critère, mettez en paramètre un prédicat permettant de comparer les éléments. 

list::reverse

Cette fonction renverse l'ordre des éléments, les premiers seront les derniers et les derniers seront les premiers ! Si ça peut vous être utile un jour...

list::sort !

Cette fonction trie les éléments d'un tableau !

- Soit elle ne prend aucun paramètre, l'**operator**< est alors utilisé.
- Soit un paramètre Prédicat de Comparaison que vous connaissez bien (Ben oui, vous vous rappelez quand même de notre foncteur Comparaison non ? ).

Un petit exemple même si vous avez déjà tout compris : 

Code : C++

```
struct Tri_par_vie
{
 bool operator()(const Personnage & p1, const Personnage & p2)
 {
  return p1.vie() < p2.vie();
 }
};
```

Code : C++

```
liste.sort( Tri_par_vie() );//Les parenthèses permettent de créer un
foncteur avec le constructeur par défaut
```

Une petite astuce de types !

Tous les conteneurs mettent à disposition le type des éléments qu'ils contiennent. Par exemple, un `vector<int>` contient des `int` et un `vecteur<Personnage*>` des `Personnage*`
Les conteneurs associatifs mettent également à disposition le type de la clef utilisé. Rappelez-vosu que les `std::map` contiennent des paires !
Voici comment obtenir ces types : avec les mot-clefs `value_type` et `key_type`.

Code : C++

```
vector<int> v;
vector<int>::value_type elemVec; //type int

v.push_back(elemVec);


map<string, Personnage*> m;
map<string, Personnage*>::value_type elemMap; //type pair<string,
Personnage*>
map<string, Personnage*>::key_type nom = "Bob"; //type string

elemMap.first = nom;
elemMap.second = new Personnage;

m.insert(elemMap);
```

Ce code peut paraître un peu idiot, mais en modifiant comme ceci, ce serait déjà mieux !

Code : C++

```
typedef myVec vector<int>;
typedef myMap map<string, Personnage*>;

myVec v;
myVec::value_type elemVec; //type int

v.push_back(elemVec);


myMap m;
myMap::value_type elemMap; //type pair<string, Personnage*>
mmyMap::key_type nom = "Bob"; //type string

elemMap.first = nom;
elemMap.second = new Personnage;

m.insert(elemMap);
```

Dans certains cas, il suffira juste de changer les **`typedef`**. 

Utilisation dans une classe template

Dans le cas d'une classe template, cela peut devenir nécessaire ! Imaginez une classe template qui manipule une `map` et un `vector`. Que choisirions nous comme paramètre template ?

Code : C++

```
MyMap m;
MyVec v;
Manipule<???> manip(m, v);
```

Dans cet exemple, nous pourrions mettre `Manipule<string, Personnage*, int>` et donc dans la définition :

Code : C++

```
template <typename Clef, typename ValeurM, typename ValeurV>
class Manipule
{
 Clef c; ValeurM m; ValeurV v;
 Manipule( std::map<Clef, ValeurM>&, std::vector<ValeurV> >);
}
```

Mais ceci n'est pas toujours correct car par exemple std::map peut avoir un paramètre template en plus, celui de Comparaison. Donc il faudrait modifier les paramètres template et cela deviendrait bien compliqué. Ce qu'on aimerait c'est :

Code : C++

```
Manipule<MyMap, MyVec> m;
```

Mais dans la classe template, on aimerait par exemple avoir la clef de la `map` ou le type des éléments du `vector`. Et donc c'est là qu'intervient l'astuce. 

Code : C++

```
template <typename LaMap, typename LeVec>
class Manipule
{
typedef typename LaMap::key type Clef;
typedef typename LaMap::value type ValeurM;
typedef typename LeVec::value_type ValeurV;

 Clef c; ValeurM m; ValeurV v;
 Manipule( std::map<Clef, ValeurM>&, std::vector<ValeurV> );
}
```

Observez l'obligation d'utiliser le mot clef **`typename`** pour indiquer au compilateur que nous définissons un type ! En effet, LaMap::key_type par exemple pourrait être une variable de type **`static`** ou une fonction de la classe !

Et dans notre classe, nous pourrions créer un itérateur par exemple ! 
Et les itérateurs pointent toujours vers... un élément contenu dans le conteneur ! Donc ... `value_type` !
Exemple dans la classe `Manipule` :

Code : C++

```
typedef typename LaMap::iterator IteratorM;
typedef typename LeVec::iterator IteratorV;

//On sait que les IteratorM pointeront vers ValeurM !
```

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous

Répondez dans l'ordre, s'il vous plaît

Itérateurs

Qu'affiche ce code ? (Allez on est at-ten-tif et on ne triche pas !)

Code : C++

```
vector<int> tab(3,6);
tab.resize(5,2);

int tableauC[] = {1,3,5,4,0};

vector<int>::iterator it = tab.begin();
it += 4;
advance(it, -2);

tab.insert(it, tableauC + 1, tableauC + 3);

tab.erase(tab.end() - 6);
tab.erase(tab.begin() + 2, tab.end() - 1);

cout << "(" << *(tab.end() - 2) << ")";
```

- ◯ (0)
- ◯ (1)
- ◯ (2)
- ◯ (3)
- ◯ (4)
- ◯ (5)
- ◯ (6)
- ◯ Rien de bon : Erreur !

Qu'affiche ce code ?

Code : C++

```
map<string, int> numeros;
numeros["Un"] = 1;
numeros["Deux"] = 2;
numeros["Trois"] = 3;
cout << numeros.begin()->second ;
```

- ◯ Un
- ◯ Deux
- ◯ Trois

- ◯ 1
- ◯ 2
- ◯ 3
- ◯ Rien de bon : Erreur !

Qui se présente ?

Code : C++

```
map<string, Personnage*> ennemis;
ennemis.insert("Le cogneur", new Personnage("Le cogneur"));
ennemis.insert("La manipulatrice", new Personnage("La
manipulatrice"));
ennemis.insert("Le personnage masqué", new Personnage);
(ennemis.end() - 2)->sePresenter();
```

- ◯ Le cogneur
- ◯ La manipulatrice
- ◯ Le personnage masqué
- ◯ Le personnage sans nom
- ◯ L'erreur de compilation

J'ai décidé d'utiliser la classe Duree du cours de M@teo dans une `std::map`, voici la classe (pour que vous vous en rappelez ) :

Code : C++

```
class Duree
{
    public:

    Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:

    int m_heures;
    int m_minutes;
    int m_secondes;
};
```

Voici mon utilisation, que contiendra mon objet "lesTemps" à la fin du code ?

Code : C++

```
map<Duree, string> lesTemps;
lesTemps[Duree(4,2,1)] = "Megaz";
lesTemps[Duree(5,3,6)] = "Robert";
lesTemps[Duree(5,3,6)] = "Bob";
```

- ◯ 3 éléments : Megaz, Robert, Bob
- ◯ 2 éléments : Megaz et Robert
- ◯ 2 éléments : Megaz et Bob
- ◯ 2 éléments : Robert et Bob
- ◯ Erreur !

J'ai ajouté une fonction à la classe Duree de la question suivante, voici son prototype :

Code : C++

```
bool operator<(const Duree & duree) const;
```

Voici sa définition (La même que celle du chapitre sur la surcharge d'opérateur de M@teo21) :

Code : C++

```
bool Duree::operator<(const Duree & duree) const
{
    if (m_heures < duree.m_heures)
        return true;
    else if (m_heures == duree.m_heures && m_minutes <
duree.m_minutes)
        return true;
    else if (m_heures == duree.m_heures && m_minutes ==
duree.m_minutes && m_secondes < duree.m_secondes)
        return true;
    else
        return false;
}
```

Qu'affichera ce code ?

Code : C++

```
map<Duree, string> lesTemps;
Duree t1(5,6,7), t2(5,7,6), t3(2,7,59);
lesTemps[t1] = "Hey ";
lesTemps[t2] = "Apple ! ";
lesTemps[t3] = "I'm an Orange ! ";
for(map<Duree, string>::iterator it = lesTemps.begin(); it !=
lesTemps.end(); ++it)
   cout << it->second;
```

- ◯ I'm an Orange ! Apple ! Hey
- ◯ I'm an Orange ! Hey Apple !
- ◯ Hey Apple ! I'm an Orange !
- ◯ Hey I'm an Orange ! Apple !
- ◯ Apple ! I'm an Orange ! Hey
- ◯ Apple ! Hey I'm an Orange !
- ◯ Erreur de compilation...

Avec la classe Duree définie ci-dessus avec en plus une fonction "afficher" :

Code : C++

```
void Duree::afficher()
{
    bool afficher_secondes_si_zero = true;
    if(m_heures > 0)
    {
        afficher_secondes_si_zero = false;
        cout << m_heures << " heure(s)";
    }
    if(m_minutes > 0)
    {
        afficher_secondes_si_zero = false;
        cout << m_minutes << " minute(s)";
    }
    if(m_secondes > 0 || afficher_secondes_si_zero)
    {
        cout << m_secondes << " seconde(s)";
    }
}
```

Que se passe-t-il ?

Code : C++

```
map<string, Duree> records;
records["Panda"] = Duree(5);
records["Tigre"] = Duree(0,0,30);
records["Guepard"].afficher();
```

- ◯ Affiche "Panda"
- ◯ Affiche "Tigre"
- ◯ Affiche "Guepard"
- ◯ Affiche "5 heure(s)"
- ◯ Affiche "30 seconde(s)"
- ◯ Affiche "0 seconde(s)"
- ◯ N'affiche rien du tout, mais sans erreurs
- ◯ Erreur...

Toujours avec la même classe Duree, qu'affiche ce code ? (En admettant qu'une paire affichera ses éléments entre accolades, une Duree ses valeurs (h min s) et un itérateur affiche l'élément qu'il pointe).

Code : C++

```
map<Duree, string> lesTemps;
map.insert(make_pair(Duree(0,10,20), "Jean"));
map.insert(make_pair(Duree(0,10,20), "Jean-Paul"));
map.insert(make_pair(Duree(1,10,20), "Jean-Paul-Jacques"));
cout << map.insert(make_pair(Duree(1,10,20), "Jean-Paul-Jacques en
enfer")).second;
```

- ◯ true
- ◯ false
- ◯ {1h 10min 20s, "Jean-Paul-Jacques en enfer")}
- ◯ Jean-Paul-Jacques en enfer
- ◯ 1h 10min 20s
- ◯ Jean-Paul-Jacques
- ◯ Erreur, Erreur !

Après le code de la question précédente, combien d'éléments contient la `std::map` ?

- ◯ 1
- ◯ 2
- ◯ 3
- ◯ 4
- ◯ Aucun, insert ne fonctionne pas

Cochez l'affirmation **fausse**.

- ◯ Une fonction est un foncteur
- ◯ Un prédicat est un foncteur
- ◯ Une classe sans opérateur () surchargé, qui contient des foncteurs, est une classe
- ◯ Une classe avec opérateur () surchargé crée toujours des prédicats
- ◯ Une classe avec opérateur () surchargé crée toujours des foncteurs
- ◯ Les prédicats renvoient toujours des booléens
- ◯ Si f est un objet avec lequel on peut écrire f() sans paramètres entre les parenthèses, alors celui est un foncteur.
- ◯ Le cosinus de 7pi/2 radians vaut la racine carrée de 0 au cube

Combien d'éléments contient la pile après ce code ?

Code : C++

```
stack<string> pile;
while(pile.size() < 5)
   pile.push("Blaaaa");
for(int i = 10;i > 0;i--)
   pile.pop();
```

- ◯ 15
- ◯ 5
- ◯ 1
- ◯ Aucun
- ◯ Erreur... Boucle infinie
- ◯ Erreur... Autre raison

Que renvoie la fonction `multimap::find` si la clef passée en paramètre est associée à plusieurs valeurs associées ?

- ◯ Un itérateur qui pointe vers un élément correspondant à la clef.
- ◯ Une paire "itérateur-itérateur" dont le premier itérateur pointe vers le premier élément correspondant à la clef, et le deuxième pointe vers l'élément suivant le dernier correspondant à la clef.
- ◯ L'itérateur end()
- ◯ Un tableau d'itérateurs correspondants à tous les éléments avec la clef
- ◯ Une std::multimap ne peut pas avoir plusieurs éléments avec la même clef...

En considérants ces types-ci :

Code : C++

```
typedef map<Coordonnees, Personnage*> MyMap;
typedef pair<Coordonnees, Personnage*> MyPair;
```

Quelle(s) phrase est(sont) correcte(s) parmi celles-ci ?
Un `MyMap::iterator` se comporte comme un pointeur vers...

1. ... `MyMap::value_type`
2. ... `MyPair`
3. ... `Personnage*`
4. ... `Coordonnees`

- ◯ La 1 uniquement !
- ◯ La 2 uniquement !
- ◯ La 3 uniquement !
- ◯ La 4 uniquement !
- ◯ La 1 et la 2 !
- ◯ La 1 et la 3 !
- ◯ La 2 et la 3 !

Correction !

Statistiques de réponses au QCM

Voilà je vous ai -j'espère- un peu éclairci sur les conteneurs de la STL (Surtout sur `std::map` me direz vous ! C'est normal, c'est celui que je préfère), ainsi que sur les itérateurs, les foncteurs.
J'espère que leurs utilisations rendront votre code plus clair et/ou plus efficace et que vous penserez à 2 fois avant d'utiliser un ~~tableau-c-like~~ `std::vector`.
Je rappelle également que ce schéma est très utile pour savoir quel conteneur choisir. Maintenant que vous savez les utiliser.

Si vous avez des exemples d'utilisations intéressantes, laissez un commentaire ! Si assez d'exemples sont donnés, je dédierai une nouvelle partie au tutoriel ! (Une partie "Pratique").

Je tiens quand même à citer mes sources pour ce tutoriel (Et je vous invite à y jetez un coup d'oeil, bien que j'ai fait tout un

tutoriel moi) : http://www.cplusplus.com/reference/stl/

Remarquez que la page de garde vous montre un tableau qui vous indique quelles fonctions peuvent être utilisées avec quels conteneurs.