



Piscine C++ - d16

Les conteneurs de la STL

Matthieu “Opera” COUSIN cousin_c@epitech.eu

Abstract: Ce document est le sujet du d16 : Les conteneurs de la STL

Table des matières

I	REMARQUES GÉNÉRALES	2
II	Exercice 0	3
III	Exercice 1	5
IV	Exercice 2	8
V	Exercice 3	13
VI	Exercice 4	15
VII	Exercice 5	18


Chapitre I

REMARQUES GÉNÉRALES

- LISEZ LES REMARQUES GÉNÉRALES ATTENTIVEMENT!!!!
 - Vous n'aurez aucune excuse si vous avez 0 parce que vous avez oublié une consigne générale ...
- REMARQUES GÉNÉRALES :
 - La STL est un ensemble de choses à voir. Donc faites tous les exos de la journée. Ca N'EST PAS progressif. Donc ne soyez pas stupides et faites toute la journée, la STL est un des concepts indispensables du C++
 - Les classes doivent être canoniques (Comprendre : on teste que les classes sont bien canoniques. Si ce n'est pas le cas, 0 à l'exercice).
 - Le C n'est pas d'actualité. Toutes les fonctions `[asvn]printf`, `*alloc` sont interdites.
- COMPILATION DES EXERCICES :
 - La moulinette compile votre code avec les flags : `-W -Wall -Werror`
 - Pour éviter les problèmes de compilation de la moulinette, incluez les fichiers nécessaires dans vos fichiers `include (*.h)`.
 - Notez bien qu'aucun de vos fichiers ne doit contenir de fonction `main` . Nous utiliserons notre propre fonction `main` pour compiler et tester votre code.
 - Le répertoire de rendu est : `(DÉPOT SVN - piscine_cpp_d16-promo-login_x)/exN` (N étant bien sur le numéro de l'exercice).

Chapitre II

Exercice 0

	Exercice : 00	points : 4
std : :stack		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex00		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : Parser.h, Parser.cpp		
Remarques : n/a		
Fonctions Interdites : *alloc - free - *printf		

Le conteneur `Stack` est une implémentation d'une pile LIFO.

Sa liste de fonctions membres est très limitée, et il n'est fait que pour certains types d'opérations. Les éléments sont insérés et retirés au sommet de la `stack`.

En guise d'échauffement, nous allons donc utiliser des conteneurs simples, les stacks, afin d'implémenter un parseur d'expressions mathématiques.

Vous allez coder une classe `Parser` à qui nous donnerons à manger et qui se fera une joie de nous donner le résultat. Pour cela, elle doit contenir deux stacks : une d'opérateurs et une d'opérandes.

Dernier détail, afin de vous faciliter (un peu) la tâche, les expressions seront parenthésées (ex : " $((1+2)+3)$ "). Vous n'aurez donc qu'à empiler les nombres et les opérateurs et à effectuer lorsque vous rencontrez une parenthèse fermante.

- Les nombres seront toujours positifs.
- Vous ne devez gérer que `+`, `-`, `*`, `/`, `%`.
- Les expressions seront toujours valides, on ne vous demande pas de coder un `eval_expr`.

Classe :

```
1 // Il s'agit de notre parseur. Elle doit contenir les methodes suivantes :
2 // Prend une expression en parametre et fait les operations au fur et
3 // a mesure qu'elle lit.
4 // Si la stack d'operandes n'est pas vide, on effectue une addition entre
5 // le resultat de l'expression courante et le residu.
6
7 void feed(const std::string&);
8
9 // Renvoie le resultat de l'operation
10 int result() const;
11
12 // Remet l'instance dans son etat initial.
13 void reset();
```


```
1 #include <iostream>
2 #include "Parser.h"
3
4 int main()
5 {
6     Parser p;
7
8     p.feed("((12*2)+14)");
9     std::cout << p.result() << std::endl;
10    p.feed("((17 % 9) / 4)");
11    std::cout << p.result() << std::endl;
12    p.reset();
13    p.feed("(17 - 4) * 13");
14    std::cout << p.result() << std::endl;
15    return 0;
16 }
```

Sortie :

```
1 38
2 40
3 169
```

Chapitre III

Exercice 1

	Exercice : 01	points : 3
std::vector		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex01		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : DomesticKoala.h, DomesticKoala.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Dans cet exercice, nous allons dresser des Koalas.

On va leur apprendre à réagir à certains caractères précis. Et vu que nos Koala ont grandi en notre compagnie, ils sont capables de comprendre toute la table ascii :-).

Donc vous allez coder une classe `DomesticKoala` , qui représentera un Koala capable d'apprendre à faire des choses, et de les reproduire. Toutes les petites choses que peuvent faire un Koala seront des fonctions membres d'une classe `KoalaAction`. Vous n'avez pas à rendre cette classe. Par contre je vous recommande plus que vivement de la coder pour vos tests.

Classes:

- `KoalaAction` :
 - Une classe variable, que vous allez coder pour les besoins de vos tests, et qui sera remplacée pendant le ramassage.
 - Cette classe se compose simplement d'un constructeur public par défaut, et d'une série de fonctions membres ayant toutes le prototype suivant :

```
1 void fctName(const std::string&);
```

- DomesticKoala :
 - Une classe qui possède des accesseurs pour des pointeurs sur des fonctions membres de la classe `KoalaAction` . Voici la liste des fonctions membres publiques:

```
1
2  DomesticKoala(KoalaAction&); // Constructeur principal
3  ~DomesticKoala(); // Ca vous
4  DomesticKoala(const DomesticKoala&); // devez connaitre
5  DomesticKoala& operator=(const DomesticKoala&); // a force... :-)
6
7  typedef XXXXX methodPointer_t; // A vous de trouver comment
8                                // on ecrit un
9                                // pointeur sur fonction membre.
10
11  // Permet de recuperer un vector contenant tous les pointeurs sur
12  // fonction membre
13  std::vector<methodPointer_t> const * getActions(void) const;
14
15  // Assigne un pointeur sur fonction membres, en liant le caractere
16  // (ordre), au pointeur (action).
17  void learnAction(unsigned char, methodPointer_t);
18
19  // Supprime l'ordre.
20  void unlearnAction(unsigned char);
21
22  // Effectue l'action liee a l'ordre. La string est le parametre de
23  // la fonction membre.
24  void doAction(unsigned char, const std::string&);
25
26  // Affecte une nouvelle classe KoalaAction au koala domestique.
27  // Cette action declenche une purge de la table de pointeur.
28  void setKoalaAction(KoalaAction&);
```

Notes:

- En cas d'appel à un ordre non défini, il ne se passe rien. Vous donnez un ordre à votre Koala domestique, et il se contente de vous regarder d'un air béat.
- A vous d'écrire votre propre classe `KoalaAction` pour les tests. Elle sera changée pour les tests.

Voici un main de test, qui suppose que nous avons déjà une classe `KoalaAction` avec les fonctions membres `eat()` , `sleep()` , `goTo()` , et `reproduce()`.

Contraintes:

- `DomesticKoala k;` NE DOIT PAS COMPILER


```
1 #include <iostream>
2 #include <cstdlib>
3 #include 'DomesticKoala.h'
4 #include 'KoalaAction.h'
5
6 int main(int argc, char **argv)
7 {
8     KoalaAction action;
9     DomesticKoala *dk = new DomesticKoala(action);
10
11     dk->learnAction('<', &KoalaAction::eat);
12     dk->learnAction('>', &KoalaAction::goTo);
13     dk->learnAction('#', &KoalaAction::sleep);
14     dk->learnAction('X', &KoalaAction::reproduce);
15
16     dk->doAction('>', '{EPITECH.}');
17     dk->doAction('<', 'un DoubleCheese');
18     dk->doAction('X', 'une Mouette');
19     dk->doAction('#', 'La fin de la piscine C++, et d'un Astek brulant sur un
20     bucher');
21     return 0;
22 }
```

Sortie:

```
1 Je vais a: {EPITECH.}
2 Je mange: un DoubleCheese
3 Je tente de me reproduire avec: une Mouette
4 Je dors et je reve de: La fin de la piscine C++, et d'un Astek brulant sur un
    bucher
```


Chapitre IV

Exercice 2

	Exercice : 02	points : 3
std::list		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex02		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : Event.h, Event.cpp, EventManager.h, EventManager.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Le Koala est un animal organisé. Il mange, il dort, il se reproduit, il se lave, et encore plein d'activités amusantes. Vous allez donc coder un gestionnaire d'événement en utilisant le conteneur `List`. Ce gestionnaire d'événement permettra de rajouter des actions à instant `T`, connaître l'action à effectuer à l'instant `T`, ajouter `T` unités de temps au temps actuel, et d'autres choses. Le détail se trouve au dessus de chaque fonction membre.

Deux classes sont à réaliser.

- Une classe `Event`, qui représente un événement. Un événement étant défini par le moment où il doit s'effectuer, et une chaîne qui le représente.
- Une classe `EventManager`, qui gère les événements.

classe `Event`:

```

1      Event(void);
2      Event(unsigned int, const std::string&);
3      ~Event(void);
4      Event(const Event&);
5      Event& operator=(const Event&);
6
7      unsigned int getTime(void) const;
8      const std::string& getEvent(void) const;
9      void setTime(unsigned int);

```

```
10 void setEvent(const std::string&);
```

```
1  classe EventManager:
2      EventManager();
3      ~EventManager();
4      EventManager(EventManager const &);
5      EventManager& operator=(EventManager const &);
6
7      // Ajoute un Event a la liste. Si le temps de l'evenement est
8      // inferieur au temps courant, l'Event n'est pas ajoute.
9      // Si il existe deja un/des evenement(s) au meme temps, on
10     // ajoute le nouvel element apres
11     void addEvent(const Event&);
12
13     // Supprime tous les Event a l'instant t
14     void removeEventsAt(unsigned int);
15
16     // Affiche la liste des evenement sous le format suivant:
17     // 8: Se reveiller
18     // 9: Debut de la journee
19     // 12: Manger
20     // etc...
21     void dumpEvents(void) const;
22
23     // Affiche tous les evenement a l'instant t sous la forme vue
24     // precedemment.
25     void dumpEventAt(unsigned int) const;
26
27     // Ajoute t au temps courant. Affiche la description de tous les
28     // evenements jusqu'au temps courant, et les supprime de la liste.
29     void addTime(unsigned int);
30
31     // Ajoute une liste d'evenement a la liste courante. Attention aux
32     // evenement invalides (t deja passe) et a l'ordre
33     // d'insertion :-)
34     void addEventList(std::list<Event>&);
```

Voici un main d'exemple :

```

1 //----- Debut -----
2 //
3 // Piscine C++
4 // Jour 16
5 //
6 // Ex2: STL List
7 //
8 // "A la queue-leu-leu, et sans les mains."
9 //
10
11 #include <cstdlib>
12 #include <iostream>
13 #include "EventManager.h"
14
15 int main(int argc, char **argv)
16 {
17     EventManager *em = new EventManager();
18
19     em->addEvent(Event(10, "Manger"));
20     em->addEvent(Event(12, "Finir les exos"));
21     em->addEvent(Event(12, "Comprendre le truc"));
22     em->addEvent(Event(15, "Mettre les droits"));
23     em->addEvent(Event(8, "Demander qu'est ce qu'un const_iterator"));
24     em->addEvent(Event(11, "Lavage de mains pour pas que les claviers sentent
        le grec"));
25     em->dumpEvents();
26     std::cout << "====" << std::endl;
27
28     // Suite a une ingestion massive de feuilles d'eucalyptus avariees, les
29     // exos du jour sont annules.
30     em->removeEventsAt(12);
31
32     em->dumpEvents();
33     std::cout << "====" << std::endl;
34
35     // C'est pas tout mais le temps passe vite
36     em->addTime(10);
37     std::cout << "====" << std::endl;
38
39     em->dumpEvents();
40     std::cout << "====" << std::endl;
41
42     // Suite a l'ingestion susnommee et pour pas que vous perdiez la main,
43     // une serie d'exos sont ajoutes.
44     std::list<Event> todo;
45     todo.push_front(Event(19, "La vengeance du Koala"));
46     todo.push_front(Event(20, "Le retour de la vengeance du Koala"));
47     todo.push_front(Event(21, "Le come back du retour de vengeance du Koala"));
48     todo.push_front(Event(22, "La suite du come back du retour de la vengeance

```


```
        du Koala"));
49     todo.push_front(Event(9, "Comment ca c'est a faire pour ce matin ?"));
50     todo.push_front(Event(1, "Non la c'est abuse quand meme..."));
51
52     em->addEventList(todo);
53
54     em->dumpEvents();
55     std::cout << "====" << std::endl;
56
57     // J'ai oublie un truc, mais quoi ???
58     em->dumpEventAt(15);
59     std::cout << "====" << std::endl;
60
61     // Et on finit la journee dans la joie et la bonne humeur.
62     em->addTime(14);
63
64     return 0;
65 }
66
67 //----- Fin -----
```

Et voici la sortie:

```
1 8: Demander qu'est ce qu'un const_iterator
2 10: Manger
3 11: Lavage de mains pour pas que les claviers sentent le grec
4 12: Finir les exos
5 12: Comprendre le truc
6 15: Mettre les droits
7 =====
8 8: Demander qu'est ce qu'un const_iterator
9 10: Manger
10 11: Lavage de mains pour pas que les claviers sentent le grec
11 15: Mettre les droits
12 =====
13 Demander qu'est ce qu'un const_iterator
14 Manger
15 =====
16 11: Lavage de mains pour pas que les claviers sentent le grec
17 15: Mettre les droits
18 =====
19 11: Lavage de mains pour pas que les claviers sentent le grec
20 15: Mettre les droits
21 19: La vengeance du Koala
22 20: Le retour de la vengeance du Koala
23 21: Le come back du retour de vengeance du Koala
24 22: La suite du come back du retour de la vengeance du Koala
25 =====
26 15: Mettre les droits
27 =====
28 Lavage de mains pour pas que les claviers sentent le grec
29 Mettre les droits
30 La vengeance du Koala
31 Le retour de la vengeance du Koala
32 Le come back du retour de vengeance du Koala
33 La suite du come back du retour de la vengeance du Koala
```

Chapitre V

Exercice 3

	Exercice : 03	points : 3
std::map		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex03		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : BF_Translator.h, BF_Translator.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Dans cet exercice, vous allez créer un traducteur BrainFuck.

Après ces deux dures semaines à naviguer dans le brouillard sur cette mer houleuse qu'est la programmation objet, vous allez goûter au plaisir d'un langage impératif simple et fonctionnel.

Ce langage a pour nom BrainFuck.
BrainFuck pour "Ouverture d'esprit" ("Brain" == "Esprit", "Fuck" == "Ouverture").
Et si vous ne me croyez pas vous avez tort :-D

Le langage BrainFuck est composé de 8 instructions qui permettent de refaire le monde en mieux.

Le schéma d'exécution est le suivant :

Le programme démarre avec un tableau alloué de 60Ko, et un pointeur sur le premier élément du tableau.

Vous avez alors les 8 instructions suivantes pour agir:

- '+' → Incrémente l'octet pointé
- '-' → Décrément l'octet pointé
- '>' → Avance le pointeur d'un octet

- '<' → Recule le pointeur d'un octet
- '?' → Affiche l'octet courant sur la sortie standard
- ',' → Lit un caractère sur l'entrée standard et le stocke dans l'octet courant
- '[' → Saute à l'instruction apres le ']' correspondant si l'octet pointe vaut 0
- ']' → Saute au '[' correspondant

Vous devez donc écrire une classe `BF_Translator`, qui possédera une fonction membre :

```
int      translate(std::string in, std::string out)
```


qui traduira le fichier BrainFuck “in” et écrira le code C correspondant dans le fichier “out”.

Cette fonction membre retournera une valeur d'erreur ($\neq 0$) en cas de pépin. (Typiquement, impossible d'ouvrir le fichier source ou le fichier destination)

Vous devez utiliser les `map`, les `string`, et les `fstream`. Du C++ quoi...

Chapitre VI

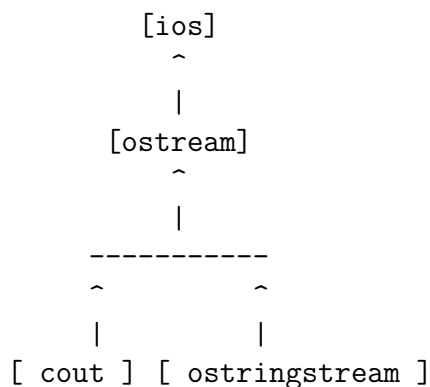
Exercice 4

	Exercice : 04	points : 3
std::ostream		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex04		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : Ratatouille.h, Ratatouille.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

`ostream` est une interface permettant d'utiliser des chaînes comme si il s'agissait d'un flux.

Vous êtes maintenant familier de l'utilisation du flux `std::cout`, et vous avez compris tout l'intérêt qu'on peut y trouver.

Donc vous avez déjà tout compris à `ostream`. Un petit schéma vaut mieux qu'un long discours :



`cout` vous permet d'envoyer des types de base et des objets sur un flux, qui est concrètement la sortie standard.

`ostream` vous permet d'envoyer des types de base et des objets sur un flux, qui est concrètement une chaîne.

Après une dure journée de piscine, il se fait faim.
Vous allez donc coder une classe `Ratatouille`, qui permet d'ajouter tout un tas d'ingrédients de nature différente dans la marmite et d'extraire après tous ces ajouts un plat succulent qui est la combinaison de tous les ingrédients.

Méthodes:

```
1 // La classe canonique
2 Ratatouille();
3 Ratatouille(Ratatouille const &);
4 ~Ratatouille();
5 Ratatouille &operator=(const Ratatouille &);
6
7 // Les fonctions membres pour ajouter des ingredients dans la marmite
8 Ratatouille &addVegetable(unsigned char);
9 Ratatouille &addCondiment(unsigned int);
10 Ratatouille &addSpice(double);
11 Ratatouille &addSauce(const std::string &);
12 // La fonction membre pour extraire le resultat
13 // La chaine renvoyee est la concatenation de tous les ingredients
14 // ajoutes.
15 std::string kooc(void);
```

Exemple:


```
1 int main()
2 {
3     Ratatouille rata;
4
5     rata.addSauce('Tomate').addCondiment(42).addSpice(123.321);
6     rata.addVegetable('x');
7
8     std::cout << "On goute: " << rata.kooc() << std::endl;
9
10    rata.addSauce('Bolognaise');
11    rata.addSpice(3.14);
12
13    std::cout << "Vas-y goute moi ca: " << rata.kooc() << std::endl;
14
15    // Attend vas-y donne ta marmite j'en prend un peu et j'essaie un autre truc
16    Ratatouille glurp(rata);
17
18    glurp.addVegetable('p');
19    glurp.addVegetable('o');
20    glurp.addSauce('Tartare');
21
22    std::cout << "Et maintenant: " << glurp.kooc() << std::endl;
23
24    // OK je prend c'est bon...
25    rata = glurp;
26    std::cout << "Je regoute c'est trop booon: " << rata.kooc() << std::endl;
27    return 0;
28 }
```

Produit la sortie suivante:

```
1 koala@arbre$ ./rata
2 On goute: Tomato42123.321x
3 Vas-y goute moi ca: Tomato42123.321xBolognaise3.14
4 Et maintenant: Tomato42123.321xBolognaise3.14poTartare
5 Je regoute c'est trop booon: Tomato42123.321xBolognaise3.14poTartare
6 koala@arbre$
```

Chapitre VII

Exercice 5

	Exercice : 05	points : 5
Pile Difforme		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d16-promo-login_x)/ex05		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : MutantStack.hpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Vous vous rappelez des **stacks** de l'exercice 0 ?
 Une particularité du conteneur **stack** est qu'il est l'un des seuls conteneurs de la STL à ne pas être itérable.
 Donc... nous allons l'itérer afin de réparer cette injustice.

Vous allez devoir implémenter une classe nommée **MutantStack** , qui possédera toutes les fonctions membres du conteneur **stack** , plus la possibilité de créer et d'utiliser des itérateurs.



Tout include d'un autre conteneur que **stack** est RIGOREUSEMENT INTERDIT !

Les opérations suivantes devront être possible:

```
1 int main()
2 {
3     MutantStack<int> mstack;
4
5     mstack.push(5);
6     mstack.push(17);
7
8     std::cout << mstack.top() << std::endl;
9
10    mstack.pop();
11
12    std::cout << mstack.size() << std::endl;
13
14    mstack.push(3);
15    mstack.push(5);
16    mstack.push(737);
17    [...]
18    mstack.push(0);
19
20    MutantStack<int>::iterator it = mstack.begin();
21    MutantStack<int>::iterator end_it = mstack.end();
22
23    ++it;
24    --it;
25    while (it != end_it)
26    {
27        std::cout << *it << std::endl;
28        ++it;
29    }
30    std::stack<int> s(mstack);
31    return 0;
32 }
```

Le comportement de sortie doit être le même que si on remplaçait `MutantStack` par `list` par exemple...

Enjoy!