C++ Pool - d16

The STL containers

Clément “Opera” COUSIN cousin_c@epitech.eu
Adrien AUBEL aubel_a@epitech.eu

Abstract: This document is the subject of d16: The STL containers

Contents

Chapter I

GENERAL RULES

- READ THE GENERAL RULES CAREFULLY!!

  You will have no possible excuse if you end up with a 0 because you didn't follow one of the general rules

- GENERAL RULES:

  - STL is something you need to see. So do all the exercises today. This IS NOT progressive. So don't be stupid and do everything this day, the STL is one of the C++'s most vital concepts.
  - Classes have to be canonical (understand: we will test that, and if your classes are not, your exercise score will be 0).
  - The C language is not today's topic. All `[asvn]printf, *alloc` functions are forbiden.

- COMPILATION OF THE EXERCISES:

  - The Koalinette compiles your code with the following flags : `-W -Wall -Werror`
  - To avoid compilation problems with the Koalinette, include every required headers in your headers.
  - Note that none of your files must contain a `main` function. We will use our own to compile and test your code.
  - This subject may be modified up to 4h before turn-in time. Refresh it regularly !
  - The turn-in dirs are `(SVN REPOSITORY - piscine_cpp_d16-promotion-login_x/exN)` , N being the exercise number

Chapter II

Exercise 0

Exercise : 00	points : 4
std::stack	
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex00`	
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`
Makefile: `No`	Rules: `n/a`
Files to turn in : `Parser.h, Parser.cpp`	
Remarks : `n/a`	
Forbidden functions : `*alloc - free - *printf`	

The `Stack` container is a LIFO stack implementation.

Its member functions number is very limited, since it's designed for only a few operation types. Elements are inserted and popped from the top of the `Stack` .

As a warm-up, we are going to use simple containers, the stacks, in order to implement a mathematical expressions parser.

You are now going to create a `Parser` class. We will feed it and it will enjoy giving us the result. This class will contain two stacks: an operators one, and an operands one.
Last detail, for making your task (a little) easier, expressions will be parenthesized (for example: "((1,2),3)"). So you will only need to stack up numbers and operators until a closing parenthesis is encountered.

- Numbers will always be greater than 0.
- The operators that have to be handled are +, -, *, /, %.
- Expressions will always be valid.

Class:

```
// This is our parser. It needs to contain the following methods:

// Takes an expression in parameter and computes the result as it reads the
   expression.
// If the operands stack isn't empty, we compute an addition between the
   current expression result and the remaining numbers.
void feed(const std::string &);

// Returns the computed result
int result() const;

// Resets the instance in its initial state.
void reset();
```

```
#include <iostream>
#include ``Parser.h''

int main()
{
    Parser p;

    p.feed(``((12*2)+14)'');
    std::cout << p.result() << std::endl;
    p.feed(``((17 % 9) / 4)'');
    std::cout << p.result() << std::endl;
    p.reset();
    p.feed(``(17 - 4) * 13'');
    std::cout << p.result() << std::endl;
    return 0;
}
```

Associated output:

```
38
40
169
```

Chapter III

Exercise 1

	Exercise : 01	points : 3
std::vector		
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex01`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `DomesticKoala.h, DomesticKoala.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

In this exercise, we're going to tame Koalas.

They will be able to handle specific characters. As they will have grown up with us, they will understand the full ASCII table :-).

So you will create a `DomesticKoala` class, which will picture a Koala able to learn to do things. Every single thing that Koalas can do will be `DomesticKoala` member functions. You don't have to turn in this class, but we recommend you to create one for testing purposes.

Classes:

- `KoalaAction` :
  - A custom class, created for your tests, that will be erased in your turn-in.
  - This class contains a public default constructor, and several member functions all compliant with the following signature:

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

- `DomesticKoala` :
  - A class that has getters for member function pointers on the `KoalaAction` class. Here is a list of these public member functions:

```

DomesticKoala(KoalaAction&); // Main constructor
    ~DomesticKoala(); // You should
    DomesticKoala(const DomesticKoala&); // know that
    DomesticKoala& operator=(const DomesticKoala&); // at the end...
        :-)

    typedef XXXXX methodPointer_t; // You have to find how
                                   // this type is defined
                                   // as a member function pointer.

    // Retrieves a vector containing all the member function pointers
    std::vector<methodPointer_t> const * getActions(void) const;

    // Sets a member function pointer, linking the character (the
        command) to the pointer (the action).
    void learnAction(unsigned char, methodPointer_t);

    // Deletes the command.
    void unlearnAction(unsigned char);

    // Executes the action linked to the given command. The string is
        the parameter given to the member function.
    void doAction(unsigned char, const std::string&);

    // Affects a new KoalaAction class to the domestic Koala.
    // This erases the pointers table.
    void setKoalaAction(KoalaAction&);
```

Notes:

- In the event we call an undefined command, nothing happens. You give this order to your domestic Koala, and he simply looks at you, without making a single move.
- Once again, you will write your own `KoalaAction` for your tests, but it will be changed by our own at the correction.

Here is a test `main` function, using a `KoalaAction` class containing the member functions `eat` , `sleep` , `goTo` and `reproduce` .

Constraints:

- `DomesticKoala k;` SHOULD NOT COMPILE

```
#include <iostream>
#include <cstdlib>
#include ‘‘DomesticKoala.h’’
#include ‘‘KoalaAction.h’’

int main(int argc, char **argv)
{
    KoalaAction action;
    DomesticKoala *dk = new DomesticKoala(action);

    dk->learnAction(’<’, &KoalaAction::eat);
    dk->learnAction(’>’, &KoalaAction::goTo);
    dk->learnAction(’#’, &KoalaAction::sleep);
    dk->learnAction(’X’, &KoalaAction::reproduce);

    dk->doAction(’>’, ‘‘{EPITECH.}’’);
    dk->doAction(’<’, ‘‘a DoubleCheese’’);
    dk->doAction(’X’, ‘‘a Seagull’’);
    dk->doAction(’#’, ‘‘The end of the C++ pool, and an Astek burning on a
        stake’’);
    return 0;
}
```

Output:

```
I go to: {EPITECH.}
I eat: a DoubleCheese
I attempt to reproduce with: a Seagull
I sleep, and dream of: The end of the C++ pool, and an Astek burning on a
    stake
```

Chapter IV

Exercise 2

	Exercise : 02	points : 3
std::list		
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex02`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Event.h, Event.cpp, EventManager.h, EventManager.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Koalas are organized animals. They eat, sleep, reproduce, wash themselves, and do so many other funny activities. So you are going to create an event handler using the `List` container. This event handler will allow you to add actions at a `T` time, know the action to do at a `T` time, add `T` time units at the current time, and other things. Below are detailed each member function of the two classes you will have to make.

- An `Event` class, that will represent an event, which is defined by the time it has to be done, and a string representing what has to be done.
- An `EventManager` class, which will allow to handle the events.

classe Event:

```
        Event(void);
        Event(unsigned int, const std::string&);
        ~Event(void);
        Event(const Event&);
        Event& operator=(const Event&);

        unsigned int getTime(void) const;
        const std::string& getEvent(void) const;
        void setTime(unsigned int);
```

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

```
classe EventManager:
    EventManager();
    ~EventManager();
    EventManager(EventManager const &);
    EventManager& operator=(EventManager const &);

    // Adds an Event to the list.
    // If the event time is prior to current time, the event is not added.
    // If there is already an event at this time, the new event is added
        after it.
    void addEvent(const Event&);

    // Deletes all the Event occuring at T time
    void removeEventsAt(unsigned int);

    // Shows the events list using the following format:
    // 8: Wake up
    // 9: Day start
    // 12: Eat
    // etc.
    void dumpEvents(void) const;

    // Shows all the events occuring at T time, using the above format
    void dumpEventAt(unsigned int) const;

    // Adds t time units to the current time.
    // Displays the description of all events that occured between the
        previous
    // current time and the new one, and deletes these events from the list
        .
    void addTime(unsigned int);

    // Ajoute une liste d'evenement a la liste courante. Attention aux
    // evenement invalides (t deja passe) et a l'ordre
    // d'insertion :-)
    // Adds an event list to the current list.
    // Be careful to invalid events (occuring prior to current time) and to
         the insertion order.
    void addEventList(std::list<Event>&);
```

Below is a sample main:

```

#include <cstdlib>
#include <iostream>
#include "EventManager.h"

int main(int argc, char **argv)
{
    EventManager *em = new EventManager();

    em->addEvent(Event(10, "Eat"));
    em->addEvent(Event(12, "Finish the exercises"));
    em->addEvent(Event(12, "Understand the thing"));
    em->addEvent(Event(15, "Set the rights"));
    em->addEvent(Event(8, "Ask what the hell a const_iterator is"));
    em->addEvent(Event(11, "Wash my hands so that my keyboard doesn't smell
        like kebab"));
    em->dumpEvents();
    std::cout << "=====" << std::endl;

    // Following a massive rotten leaves of eucalyptus ingestion,
    // all the exercises of the day are canceled.
    em->removeEventsAt(12);

    em->dumpEvents();
    std::cout << "=====" << std::endl;

    // Hey, the time is flying!
    em->addTime(10);
    std::cout << "=====" << std::endl;

    em->dumpEvents();
    std::cout << "=====" << std::endl;

    // Following the aforementioned ingestion and to help you improve your
        skill level,
    // an exercises serie will be added.
    std::list<Event> todo;
    todo.push_front(Event(19, "The vengeance of the Koala"));
    todo.push_front(Event(20, "The return of the vengeance of the Koala"));
    todo.push_front(Event(21, "The come back of the return of the vengeance of
        the Koala"));
    todo.push_front(Event(22, "The sequel to the come back of the return of ...
         well, you get the drift."));
    todo.push_front(Event(9, "What the hell do you mean 'It's due this morning'
         ?!"));
    todo.push_front(Event(1, "No, no, you're pushing it now..."));

    em->addEventList(todo);

    em->dumpEvents();
    std::cout << "=====" << std::endl;

    // I forgot something, but what??
    em->dumpEventAt(15);
    std::cout << "=====" << std::endl;

    // And we finish the day with joy and good humour.
    em->addTime(14);

    return 0;
}
```

Here is the associated output

```
8: Ask what the hell a const_iterator is
10: Eat
11: Wash my hands so that my keyboard doesn’t smell like kebab
12: Finish the exercises
12: Understand the thing
15: Set the rights
=====
8: Ask what the hell a const_iterator is
10: Eat
11: Wash my hands so that my keyboard doesn’t smell like kebab
15: Set the rights
=====
Ask what the hell a const_iterator is
Eat
=====
11: Wash my hands so that my keyboard doesn’t smell like kebab
15: Set the rights
=====
11: Wash my hands so that my keyboard doesn’t smell like kebab
15: Set the rights
19: The vengeance of the Koala
20: The return of the vengeance of the Koala
21: The come back of the return of the vengeance of the Koala
22: The sequel to the come back of the return of ... well, you get the drift.
=====
15: Set the rights
=====
Wash my hands so that my keyboard doesn’t smell like kebab
Set the rights
The vengeance of the Koala
The return of the vengeance of the Koala
The come back of the return of the vengeance of the Koala
The sequel to the come back of the return of ... well, you get the drift.
```

Chapter V

Exercise 3

	Exercise : 03	points : 3
std::map		
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex03`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `BF_Translator.h, BF_Translator.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

In this exercise, you will create a BrainFuck translator.

After two hard weeks spent sailing in the fog in this rough sea which is the object-oriented programming, you will taste to the flavor of a simple and functional imperative language.

This language is named BrainFuck.

The BrainFuck languaged is composed of 8 instructions that allow you to make the world better.
The execution scheme is the following:
The program starts with an allocated 60 KB array, and a pointer on the start of this array.

You then have the 8 following instructions to handle:

- '+' → Increments the pointed byte
- '-' → Decrements the pointed byte
- '>' → Moves the pointer forwards by 1 byte
- '<' → Moves the pointer backwards by 1 byte

- '.' → Displays the current byte on the standard output
- ',' → Reads a character on the standard input and sets the current byte to this value
- '[' → Jumps to the instruction following the matching ']' if the current byte is set to 0
- ']' → Jumps to the matching '['

You must write a `BF_Translator` classe, that will have one member function:

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

This member function will translate the `in` BrainFuck file and will write the associate C code in the `out` file.
This member function will return 0 if successful, another value if an error occured (for example if we couldn't open the input file or write to the destination)

You must use `std::map` , `std::string` and `std::fstream` . This is C++ after all...

Chapter VI

Exercise 4

Exercise : 04	points : 3
std::ostrinstream	
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex04`	
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`
Makefile: `No`	Rules: `n/a`
Files to turn in : `Ratatouille.h, Ratatouille.cpp`	
Remarks : `n/a`	
Forbidden functions : `None`	

`ostringstream` is an interface allowing to use strings as if they were streams.
You now perfectly know how to use the `std::cout` stream, so you understand the advantages using streams.
You already know everything about `ostringstream` . A picture says more than a thousand words:

```
          [ios]
            ^
            |
         [ostream]
            ^
            |
       -----------
       ^         ^
       |         |
  [ cout ] [ ostringstream ]
```

`cout` allows you to send objects to the stream, which is really the standard output.
`ostringstream` allows you to send objects to a stream, which is really a string.

After a hard pool day, you are hungry.
You are going to create a `Ratatouille` class, with which we will add different ingredients in a cooking pot, and will extract after that a succulent dish, that will be the combination of all these ingredients.

Methods:

```
        // Canonical class
        Ratatouille();
        Ratatouille(Ratatouille const &);
        ~Ratatouille();
        Ratatouille &operator=(const Ratatouille &);

        // Member functions allowing to add ingredients in the cooking pot
        Ratatouille &addVegetable(unsigned char);
        Ratatouille &addCondiment(unsigned int);
        Ratatouille &addSpice(double);
        Ratatouille &addSauce(const std::string &);

        // The member function to extract the dish.
        // The result will be the concatenation of all the added ingredients.
        std::string kooc(void);
```

Example:

```
int main()
{
   Ratatouille rata;

   rata.addSauce(‘‘Tomato’’).addCondiment(42).addSpice(123.321);
   rata.addVegetable(’x’);

   std::cout << ‘‘We taste: ‘‘ << rata.kooc() << std::endl;

   rata.addSauce(‘‘Bolognese’’);
   rata.addSpice(3.14);

   std::cout << ‘‘C’mon, taste that: ‘‘ << rata.kooc() << std::endl;

   // C’mon, gimme your pot, i’ll just take a bit of it and try something else
   Ratatouille glurp(rata);

   glurp.addVegetable(’p’);
   glurp.addVegetable(’o’);
   glurp.addSauce(‘‘Tartar’’);

   std::cout << ‘‘And now: ‘‘ << glurp.kooc() << std::endl;

   // Looks good ...
   rata = glurp;
   std::cout << ‘‘I’ll taste again, it’s sooo good: ‘‘ << rata.kooc() << std::
      endl;
   return 0;
}
```

Here is the expected result:

```
koala@arbre$ ./rata
We taste: Tomato42123.321x
C’mon, taste that: Tomato42123.321xBolognese3.14
And now: Tomato42123.321xBolognese3.14poTartar
I’ll taste again, it’s sooo good: Tomato42123.321xBolognese3.14poTartar
koala@arbre$
```

Chapter VII

Exercise 5

	Exercise : 05	points : 5
Mutant Stack		
Turn-in directory: `(SVN REPOSITORY - piscine_cpp_d16-promo-login_x)/ex05`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `MutantStack.hpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Do you remember the `stacks` as seen in the exercise 0?
One special aspect of the `stack` container is that it is one of the only STL containers that is not iterable.
So... We will add this ability to the `stack` container to undo this injustice.

You will have to implement a class named `MutantStack`, that will have all the `stack` container member functions, only this class will be iterable.

> ⚠ `The only container that you are able to include to your exercise is stack.`

The following operations must be available:

```
int main()
{
    MutantStack<int> mstack;

    mstack.push(5);
    mstack.push(17);

    std::cout << mstack.top() << std::endl;

    mstack.pop();

    std::cout << mstack.size() << std::endl;

    mstack.push(3);
    mstack.push(5);
    mstack.push(737);
    [...]
    mstack.push(0);

    MutantStack<int>::iterator it = mstack.begin();
    MutantStack<int>::iterator end_it = mstack.end();

    ++it;
    --it;
    while (it != end_it)
    {
        std::cout << *it << std::endl;
        ++it;
    }
    std::stack<int> s(mstack);
    return 0;
}
```

The output must be the same as if we would replace `MutantStack` by `std::list` for example...

Enjoy!