



B3 - C++ Pool

B-PAV-242

Day 16

The StandarT Library



KOALA

42.0



Day 16

binary name: no binary
group size: 1
repository name: cpp_d16
repository rights: ramassage-tek
language: C++



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).



GENERAL SETPOINTS

READ THESE CAREFULLY

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



If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercises because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.



Read the examples **CAREFULLY**. They might require things that weren't mentioned in the subject...



All output goes to the standard output and must be ended with a newline character, unless specified otherwise.



Remember: you're coding in C++ now, and not in C. Therefore, the following functions are **FORBIDDEN** and their use will be punished by a -42, no questions asked:

*alloc
*printf
free



Any use of the `friend` keyword will result in a -42



You are not allowed to use any library other than the C++ standard library.



It must be possible to include each of your header files independently from the others. Headers must include all their dependencies.



All your header files will be included in the correction `main`.



None of your files must contain a `main` function



THINK. Please.



THINK



T.H.I.N.K.! For Pony!



To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that none of your files must contain a `main` function, unless specified otherwise. We will use our own `main` functions to compile and test your code.



This subject may be modified up to one hour before turn-in time!



BEFORE WE BEGIN

The **STL** is the **most important tool you will use in C++**. It is of the utmost importance that you do these exercises, watch the day's video lectures and read even a little bit of documentation. If you had to remember a single day of this **Pool**, it should be today.

By paying attention and learning a few of the tools available for you to use in the STL, you will spare yourself **hours** of work.



UNIT TESTS

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the “**How to write Unit Tests**” document on the intranet, available [here](#).

Create a directory named `tests`. For each of the classes you turn in, create a file in that directory named `tests-CLASS-NAME.cpp` containing all the tests needed to cover all of the class' possible cases (regular or irregular).

Here is a sample set of unit tests for the **string** class:

```
#include <riterion/criterion.h>
```

```
Test(string, default_value)
{
    std::string s;
    cr_assert_eq(s, "");
}
```

```
Test(string, assign)
{
    std::string s;


    s = "test";
    cr_assert_eq(s, "test");
}
```

```
Test(string, append)
{
    std::string s("test");

    s += "ing";
    cr_assert_eq(s, "testing");
}
```



EXERCISE 0 - stack

	Exercise: 00	points : 4
'std::stack'		
Turn-in directory: cpp_d16/ex00		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Parser.h, Parser.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

The `stack` class template in the STL is a **LIFO** stack implementation.

Its member functions are very limited, as it is designed to allow for a small set of operations. Elements are inserted and popped from the top of the stack.

As a warm-up, you will have to use stacks, which are simple containers, in order to implement a mathematical expression parser. This exercise is sometimes called the **20 minute challenge** in computer science, as implementing a calculator should not take more than 20 minutes.

Create a `Parser` class. We will feed it, and it will be happy to give us the result of the expression. The class will hold two stacks: one of **operators**, and another of **operands**.

To make things (a little) easier for you, input expressions will always be parenthesized (for example: `"((1,2),3)"`).

- Numbers will always be greater than 0
- The operators you have to handle are +, -, *, / and %
- Expressions will always be valid



Here is the class' description:

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

// Takes an expression as 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 to its initial state.
void reset();
```

Here is a sample main function and its expected output:

```
#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;
}
```

main.cpp

```
Terminal
~/B-PAV-242> ./a.out
38
40
169
```




EXERCISE 1 - vector

	Exercise: 01	points : 3
'std::vector'		
Turn-in directory: cpp_d16/ex01		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: DomesticKoala.h, DomesticKoala.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

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

They will be able to deal with specific characters. As they have been learning with you ever since you started your studies, they can understand the full **ASCII table**.

Create a `DomesticKoala` class which will represent a Koala that can learn to do things. Everything a Koala can do will be a `KoalaAction` member function.

You don't have to turn in this class, but we recommend to create one for testing purposes:

- `KoalaAction`
 - A custom class, created for your tests, that will be erased in your turn-in
 - Has a public default constructor and several member functions matching the `void (const std::string &) signature`
- `DomesticKoala`
 - A class with getters for `KoalaAction` member function pointers.

```
DomesticKoala(KoalaAction &);
~DomesticKoala();
DomesticKoala(const DomesticKoala &);
DomesticKoala &operator=(const DomesticKoala&);

using methodPointer_t = XXXXX; // Figuring out the actual type is up to you

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

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

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

// Executes the action linked to the given command. The string is the parameter
```



```
// given to the member function.
void doAction(unsigned char command, const std::string &param);

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

Notes:

- If an undefined command is called, nothing happens. You give the Koala the order and it simply looks at you, not moving.
- DomesticKoala k; **SHOULD NOT COMPILE**

Here is a sample `main` function using a `KoalaAction` class containing the `eat`, `sleep`, `goTo` and `reproduce` member functions, along with its expected output.

```
#include <iostream>
#include <cstdlib>
#include "DomesticKoala.h"
#include "KoalaAction.h"

int main()
{
    KoalaAction action;
    DomesticKoala dk(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;
}
```

main.cpp

```
Terminal
~/B-PAV-242> ./a.out
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
```



EXERCISE 2 - list

	Exercise: 02	points : 3
'std::list'		
Turn-in directory: cpp_d16/ex02		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Event.h, Event.cpp, EventManager.h, EventManager.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

Koalas are organized animals. They eat, sleep, reproduce, wash, and do many other funny activities. To make sure this all goes well, you are going to create an event handler using an `std::list`. This event handler will let you schedule actions at a `T` time, know the action scheduled at a given time, add `T` time units to the current time, and many other things.

Implement:

- An `Event` class representing an event, defined by the time it is scheduled for and a `string` representing what has to be done

```
Event();
Event(unsigned int time, const std::string &event);
~Event();
Event(const Event &other);
Event &operator=(const Event &rhs);

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



- An EventManager class, managing Events.

```
EventManager();
~EventManager();
EventManager(EventManager const &other);
EventManager &operator=(EventManager const &rhs);

// 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 &e);

// Deletes all the Events occurring at T time
void removeEventsAt(unsigned int time);

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

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

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

// Adds an event list to the current list.
void addEventList(std::list<Event> &events);
```



Here is a sample `main` function and its expected output.

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

int main()
{
    EventManager em;

    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;
}
```

main.cpp

```
Terminal
~/B-PAV-242> ./a.out
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.
```



EXERCISE 3 - map

	Exercise: 03	points : 3
'std::map'		
Turn-in directory: cpp_d16/ex03		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: BF_Translator.h, BF_Translator.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

In this exercise, you will create a **BrainFuck** translator.

After two long weeks sailing in the fog through the rough sea of Object-Oriented Programming, you will finally enjoy the sweet taste of a simple and functional imperative language: **BrainFuck**.

The BrainFuck language is composed of 8 instructions that let you make the world a better place. The execution scheme is the following:

The program starts with an allocated 60KB array and a pointer to the start of this array. It can then execute the following instructions:

- + -> Increments the pointed byte
- - -> Decrements the pointed byte
- > -> Moves the pointer forward by 1 byte
- < -> Moves the pointer backward by 1 byte
- . -> Prints the pointed byte to the standard output
- , -> Reads a character from the standard input and sets the pointed byte to its value
- [-> Jumps to the instruction following the matching] if the pointed byte is set to 0
-] -> Jumps to the matching [

You must write a `BF_Translator` class with a single member function:


```
int translate(const std::string &in, const std::string &out);
```

This member function will translate the `in` BrainFuck file and write the associated **C** code in the `out` file. It must return 0 if it succeeds, and any other value if an error occurs.

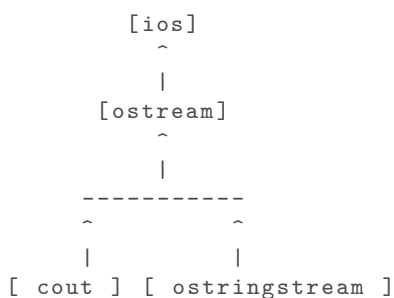
You must use `std::map`, `std::string` and `std::fstream`. This is C++ after all...



EXERCISE 4 - ostreamstringstream

	Exercise: 04	points : 3
'std::ostreamstringstream'		
Turn-in directory: cpp_d16/ex04		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Ratatouille.h, Ratatouille.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

`ostreamstringstream` is an interface that lets you use strings as if they were **streams**. You are now a professional when it comes to using `std::cout`, so you understand the advantages of using streams. You already know everything about `std::ostreamstringstream`. A picture is worth a thousand words:



After a hard day of pool, you are hungry. Create a `Ratatouille` class which we will use to add various ingredients into a cooking pot and from which we will get a delicious dish.

```
Ratatouille();
Ratatouille(Ratatouille const &other);
~Ratatouille();
Ratatouille &operator=(const Ratatouille &rhs);

// 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();
```




Here is a sample `main` function and its expected output.

```
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;
}
```

main.cpp

```
~/B-PAV-242> ./a.out
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
```



EXERCISE 5 - MUTANT STACK

	Exercise: 05	points : 5
Mutant Stack		
Turn-in directory: <code>cpp_d16/ex05</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Wextra -Werror -std=c++14</code>	
Makefile: No	Rules: <code>n/a</code>	
Files to turn in: <code>MutantStack.hpp</code>		
Notes: None		
Forbidden functions: <code>'using namespace'</code> keyword		

Do you remember the `stacks` we discussed in the first exercise? One special aspect of `std::stack` is that it is one of the few **STL containers** that are not **iterable**. Let's add this feature to fix this injustice!

Implement a `MutantStack` class with the same member functions as `std::stack` but is iterable.



The only container you are allowed to include in your exercise is `stack`.



The following code must compile:

```
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 `MutantStack` was replaced with `std::vector`.