B3 - C++ Pool

B-CPP-300

Day 16

The Standard Library

2.1

Day 16

repository name: *cpp_d16_$ACADEMICYEAR*
repository rights: ramassage-tek
language: C++

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

All your exercises will be compiled with `g++` **and the** `-Wall -Wextra -Werror` **flags**, unless specified otherwise.

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

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. It will include your header files.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.

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

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

The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++.
By the way, `friend` is forbidden too, as well as any library except the standard one.

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.

EXERCISE 0 - `stack`

Turn in: `Parser.hpp`, `Parser.cpp`

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 must 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:

```
// Takes an expression as parameter and computes the result as it reads the
    expression.
// If the operands stack isn't empty, 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:

```cpp
#include <iostream>
#include "Parser.hpp"

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;
        p.feed("(((133 / 5) + 6) * ((45642 % 127) - 21))");
        std::cout << p.result() << std::endl;
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
38
40
-35
861
```

EXERCISE 1 - `vector`

Turn in: `DomesticKoala.hpp`, `DomesticKoala.cpp`

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

They must 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 represents a Koala that can learn to do things.
Everything a Koala can do is 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, to be erased in your turn-in
It 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 &);
```

If ever 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.hpp"
#include "KoalaAction.hpp"

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;
}
```

Terminal

```
~/B-CPP-300> ./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`

Turn in: `Event.hpp`, `Event.cpp`, `EventManager.hpp`, `EventManager.cpp`

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);
```

Implement 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 occuring at T time
void removeEventsAt(unsigned int time);

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

// Shows all the events occuring 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 occured 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.hpp"

static std::list<Event> createTodoList()
{
        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 vengeance of the Koala"));
        todo.push_front(Event(22, "The sequel to the vengeance."));
        todo.push_front(Event(9, "What the hell do you mean 'this morning' ?!"));
        todo.push_front(Event(1, "No, no, you're pushing it now..."));

        return todo;
}

static populateEvents(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"));
}

int main()
{
        EventManager em;

        populateEvents(em);
        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.
        em.addEventList(createTodoList());
        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;
}
```

Terminal

```
~/B-CPP-300> ./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 vengeance of the Koala
22: The sequel to the vengeance.
=====
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 vengeance of the Koala
The sequel to the vengeance.
```

EXERCISE 3 - `map`

Turn in: `BF_Translator.hpp`, `BF_Translator.cpp`

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 simle 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:

1. the program starts with an allocated 60KB array and a pointer to the start of this array,
2. 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);
```

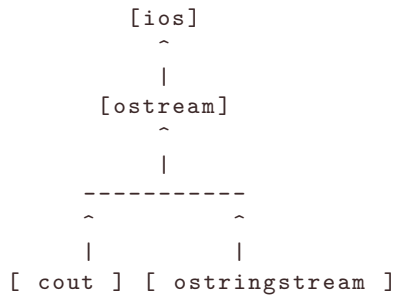
This member function translates the `in` BrainFuck file and writes the associated **C** code in the `out` file.
It returns 0 if it succeeds, and any other value if an error occurs.

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

EXERCISE 4 - ostringstream

Turn in: Ratatouille.hpp, Ratatouille.cpp

ostringstream 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::ostringstream.
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

Terminal

```
~/B-CPP-300> ./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

Turn in: `MutantStack.hpp`

Do you remember the `stacks` we discussed in the first exercise?
One specical 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`.