ECE 150 Fall 2019

Project 1

Due: 11:59pm, October 9, 2019

In this document, we will describe

1. the necessary background;
2. the functions provided by us that will be used in all the functions you implement;
3. the printing of a file, which you will have to implement;
4. the checking of a single type of delimiter, which you will have to implement;
5. the checking of the correct matching and nesting of all three delimiters, also which you will have to implement, but where we describe the algorithm and provide you with additional functions that you can use;
6. an appropriate testing approach.

We start with the background which describes the problem.

1 Background

Integrated development environments (IDEs) play an important role in helping developers write code correctly. One aspect of providing this support to developers is identifying matching delimiters. In C++ and in many other programming languages, the characters

`( ) [ ] { }`

are used to group statements and expressions. These are referred to as *parentheses*, *brackets*, and *braces*, respectively, but for more clarity and to emphasize the shape, we will sometimes refer to them as *round* parentheses, *square* brackets, and *curly* braces. We will refer collectively to such grouping characters as *delimiters*. These C++ delimiters are used in matching pairs, so `(…)`, `[…]` or `{…}`, and never in the form `)…(`, `]…[` or `}…{`, or otherwise mixed; for example, `(3, 5]` is not forbidden. Thus, we will refer to the first in any such pair, `(`, `[` or `{`, as *opening* delimiters (being either an opening parenthesis, opening bracket or opening brace) and each has a matching *closing* delimiter `)`, `]` or `}`, respectively. You will never find an opening delimiter without a matching closing delimiter in a correctly written C++ program, but it is not that simple.

In any C++ program, if you were to delete all other characters (include all characters contained in literal strings `"Hello world! :-)"` and literal characters `'}'`), you will be able to make some observations; for example, consider the following taken from two C++ source files:

Example 1:

```
(){()()(){()()()(())}(){()()(())()(())()()(())(())}{()}}
```

Example 2:

```
{{(())()(){((()))()()(){(())(){(){[][][]([][])[]}}}(()){(){[][][][]}(){}}}(){(){[]([])}()(){[]([])()}}}}
```

You will notice that there are some common features, including:

- There are as many opening delimiters of each style as there are closing delimiters.
- There does not appear to be an *overlapping* of opening and closing delimiters, e.g., `[(])`

Because these are used for grouping, these delimiters must obey the following two rules:

1. Each opening delimiter must be *matched* with the closest unmatched closing delimiter of the same style.
2. Between any matched pair of delimiters, all matched pairs of delimiters must fall between the given outer matched pair of delimiters. That is, matched pairs of delimiters must be *nested*.

Therefore, all delimiters must come in nested matching pairs of delimiters. Thus, if you look only at the delimiters in a C++ program, you may see any of the following:

```
() [] {} (()) ([]{})[] {[()]} {()[()]{[()]()[[]]}}
```

In the last example, we use color to highlight some of the matching delimiters. However, you will never see correctly written C++ programs resulting in examples such as:

`} ] )`	A closing delimiter without a matching opening delimiter.
`[} [) (] (} {] {]`	Incorrectly matched opening and closing delimiters.
`{[(`	An opening delimiter without a matching delimiter.
`{[}] {([)}]`	Unnested matched pairs of delimiters (`{[]}` and `[{}]` are valid, as are `{([])}`, `{()[]}` and `{()}[]`).

For the first example above, the following color-coded representation attempts to show the matching delimiters:

```
(){ ()()(){()()()(())} (){ ()()(())()(())()()(())(()) }{()} }
```

For the second example above, we will line up matching braces in the same column. Once it is clear that the braces are matched, you will note that all parentheses and brackets, too, are matched.

```
{
        {
                (())()() {
                        ((()))()()() {
                                ()()() {
                                        () {
                                                [][][]([][])[]
                                        }
                                }
                        }
                        (()) {
                                () {
                                        [][][][]
                                }
                                () {
                                }
                        }
                }
                () {
                        () {
                                []([])
                        }
                        ()() {
                                []([])()
                        }
                }
        }
}
```

You will write three functions that will perform operations on a given file called `test_code.cpp`. This file should be located in the same directory as your source code. You can put into this file whatever examples you want. We provide you with a default program to test that has a few errors.

2 Provided code (part 1 of 2)

Each of your functions will use the following three functions that are used to step through a file. These functions are provided in a file called `project_1.hpp`. Notice that there is an `ece150::` similar to the standard library namespace `std::` that is used for calling each of these functions. For example, `ece150::start_reading_file()` calls the function shown in 2.1.

2.1 `void ece150::start_reading_file()`

This starts reading the file `test_code.cpp` at the first character of that file. This function must be called before either of the next two functions are called.

`bool ece150::at_eof()`

This function can be called at any time, and returns true if there are no more characters in the file in question. The term "eof" is short for "end-of-file". It is an undefined operation to call this function if `ece150::start_reading_file()` has not yet been called (meaning, the behavior is undefined; it may return garbage, it may throw an error, etc.).

2.2 `char ece150::get_next_character()`

This function returns the next character in the file. It is an undefined operation to call this function if `ece150::start_reading_file()` has not yet been called or if `ece150::at_eof()` returns `true`.

3 `void print_program()`

This first function you will write simply prints the program to the screen. You will get the characters from the file one at a time and print them to the screen. The output to the screen should be similar to the file viewed in a text editor or your IDE.

4 bool does_delimiter_match(char delimiter)

In this first algorithm, we will match one of the three pairs of delimiters, either parentheses, brackets, or braces. The argument `delimiter` may be one of `'('`, `'['`, or `'{'`. If we are matching only one type of delimiter, the algorithm is straight-forward: First, if the parameter has any other value than one of the three given characters, return `false`. Otherwise, you will match only the given type of delimiter. For this, you will read one character at a time, and you will increment or decrement a counter each time you come across an opening or closing delimiter of the given type. There will, however, be circumstances where there is obviously not a matching delimiter, so you will have to print a message to the console output. While you are stepping through the file, you will be able to deduce when a closing delimiter does not have a matching opening delimiter, but only at the end of the file will you be able to deduce the number of opening delimiters that do not have closing delimiters. Thus, each time you find a closing delimiter that does not have a matching opening delimiter (it is ***unmatched***), you will print an appropriate message to the console output, and at the end you will print an appropriate message indicating the number of ***missing*** closing delimiters. The following files would give the corresponding outputs when checking for the given delimiter:

```
Hi } there
```
```
Unmatched }
```

```
Hi { there
```
```
Missing }
```

```
}{{{}{
```
```
Unmatched }
Missing }}}
```

```
)()))(()
```
```
Unmatched )
Unmatched )
Unmatched )
Missing )
```

```
[[[]]][[][]][[[]]
```
```
Unmatched ]
Missing ]
```

The following file should return `true` for all three. Adding one more delimiter or deleting one delimiter should result in one error message being printed and `false` being returned. Try adding one of each of the three kinds of delimiters and then checking for all three arguments.

```
(([()()]){}())[](){()}[[]{}()]
```

5 `bool do_delimiters_nest()`

The previous function checks if one delimiter matches, but it is also required that different matching delimiters nest appropriately, so whereas the previous function would return `true` for each of the delimiters for the input `([{)]}`, these three delimiters are clearly not nested.

We will first cover the theory we need to check not only for matching, but also for nested delimiters. We will then describe the provided functions you will use to implement the theory, and we will then describe the behavior of the required function.

51. Theory for determining nested and matching delimiters

The following is a simple algorithm that you can use to determine if the delimiters in a document are nested and matching. Start by drawing a table which you will fill from left to right, starting with the first cell:

Next, starting with the first character and scanning forward:

1. Whenever you reach an opening delimiter, place that delimiter in the next available location in your table.
2. Whenever you reach a closing delimiter, check the most recently added opening delimiter added to your table (the right-most one). If the most recently added opening delimiter matches the current closing delimiter (i.e., they are both parentheses, both brackets or both braces), then you have found a matching pair, and remove (erase) the opening delimiter from the table. There are two cases where the delimiters are not correctly balanced:
    a. You reached a closing delimiter, but the table is empty, in which case, you have an unmatched closing delimiter.
    b. The opening delimiter in the table does not match the closing delimiter, in which case, the matching delimiters are not nested.

If after you have scanned all characters in the document and no problems were found, then there are two final possibilities:

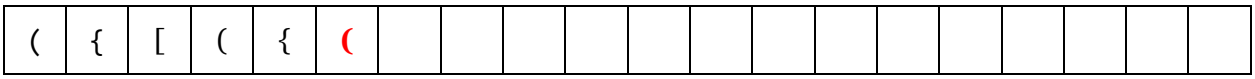
1. The table is empty, which means that all opening delimiters were found to have nested and matching closing delimiters.
2. There are still opening delimiters on the table, in which case, there are opening delimiters that do not have matching closing delimiters.

We will give three examples that lead to unmatched delimiters.

5.1.1 Example 1.

In this example, there is a closing delimiter that does not match the most recent unmatched opening delimiter that appears in our table. Thus, the (and] are both unmatched.

[ ] ({ [([ ] { (([ ]) { }] })] } [ ]) [( )]



5.1.2 Example 2.

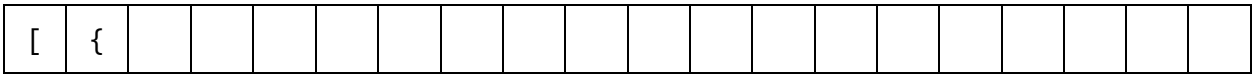
In this example, there is a closing delimiter for which there is no matching opening delimiter, so when the red character is reached, the table will be empty.

[ ] ({ [ ] } [ ])) [( )]

5.1.3 Example 3.

In this example, there are two opening delimiters for which there is no matching closing delimiter, so when the end of the sequence of characters is reached, there will be no matching delimiter for the ones indicated in red.

[ ] [({ [ ] } [ ]) { [( )]



5.2 Provided code (part 2 of 2)

To implement this algorithm, we need a way to save the delimiters into a table, and make appropriate accesses to it. We will provide you with code that builds the table and allows you to place characters onto, and erase characters from the table.

5.2.1 `void ece150::initialize_table()`

This function initializes the table you will use. You cannot call any other function unless you call this particular function first. It will set up the environment to work with this table. If you put characters into this table using the next function, but then call this function again, it will empty the table.

5.2.2 `void ece150::place_character( char delimiter )`

This places the character in the next location of the table. You should restrict yourself to placing opening delimiters onto this table. The algorithm will not work correctly, otherwise.

5.2.3 `bool ece150::is_table_empty()`

This returns `true` if the table is empty, and `false` otherwise (that is, there are still characters that can be *gotten* by the next function).

5.2.4 `char ece150::get_right_most_character()`

This returns the right-most character in the table. It is an undefined operation to call this function when the table is empty.

5.2.5 `void ece150::erase_right_most_character()`

This erases the right-most character in the table. It is an undefined operation to call this function when the table is empty.

5.3 Function specifications

Your function (`do_delimiters_nest()`) will return `true` if the delimiters both match and are nested, and `false` otherwise.

As you are reading through the characters in the file:

1. Each time you reach an opening delimiter, place that character onto the table.
2. Each time you read a closing delimiter:
    a. If that delimiter matches the right-most delimiter on the table, remove the opening delimiter from the table.
    b. If the closing delimiter does not match the right-most opening delimiter on the table, print an error message
        `Unmatched )`
    followed by an end-of-line where the `)` may be `]` or `}`, as appropriate.

If, once you are at the end of the file, the table is not empty, that means that all of the remaining entries in the table are unmatched opening delimiters. In this case, print the error message

`Missing )]}]]}}))`

where the first is the matching delimiter of the right-most character remaining on the table, the second is the matching delimiter of the next right-most character remaining in the table, and so on. This error message should be followed by an end-of-line.

The following files would give the corresponding outputs when checking for both matching and nested delimiters:

```
[(])
```

```
Unmatched ]
Missing ]
```

```
({}[()]{[[()]][
```

```
Missing ]})
```

```
{()[])}([]){[()}]}()
```

```
Unmatched )
Unmatched }
```

6 Testing strategy

Do not start with C++ code, just start by having a text file that has just one type of delimiter, then add more delimiters as you go on. To create a valid arbitrarily large structure, just start by typing two adjacent opening-closing pair:

```
[]
```

Next, randomly choose another and add it at any location, so any of:

```
()[]  [()]  []()
```

Repeat, with another randomly chosen delimiter, so any of

```
{}()[]  ({})[]  (){}[]  ()[{}]  ()[]{}  {}[()]  [{}()]  [({})]  [(){}]
           [()]{}  {}[]()  [{}]()  []{}()  [](){})  [](){}
```

and so on. Just be safe, you can start adding additional characters to make sure they are ignored:

```
[(7890-=;',./<)abcdefghijklmnopqrstuvwxyz]ABCDEFGHIJKLMNOPQRSTUVWXZY{123456}
```

Once you have a file, you can start adding or removing delimiters at random and you should expect to get errors.

NOTE: It is not cheating or plagiarism to create test files with expected outputs and to give this to your peers.

7 Marmoset Testing

All tests for this project will be secret, but the test cases you will receive will be based on the following program:

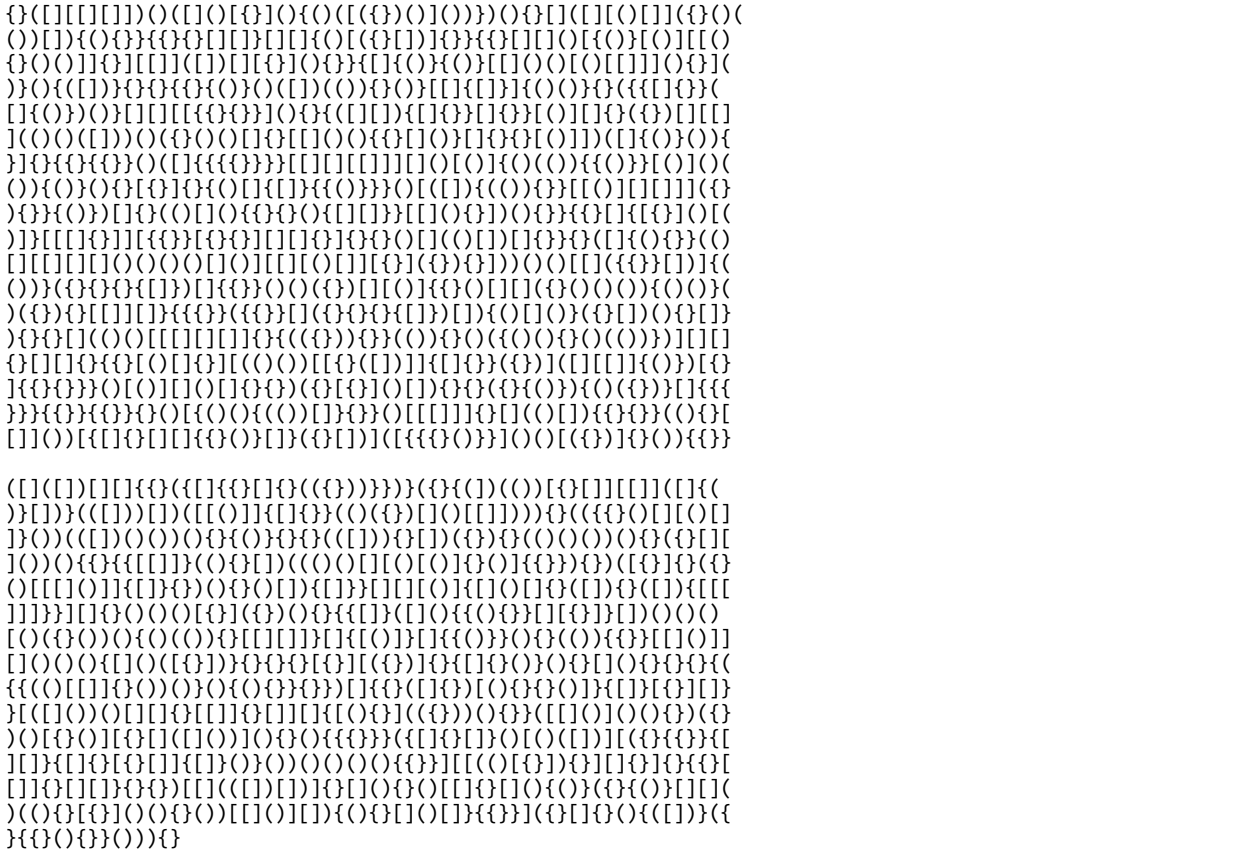
```
#include <iostream>
#include <cassert>
#include "project_1.hpp"

// Function declarations
int main();
void print_program();
bool does_delimiter_match( char del );
bool do_delimiters_nest();

// Function definitions
int main() {
        print_program();
        std::cout << does_delimiter_match( '(' ) << std::endl;
        std::cout << does_delimiter_match( '[' ) << std::endl;
        std::cout << does_delimiter_match( '{' ) << std::endl;
        std::cout << do_delimiters_nest() << std::endl;

        return 0;
}

// Your function definitions go here
```

This will read code from the test file `test_code.cpp` which is in the same directory as this file. You can populate `test_code.cpp` with whatever text you wish, but you should start by looking through the examples above, and then these will be the test cases we will give you. It is up to you to determine what the appropriate output is, and then we will test the following inputs (each block is a separate input):

```
[ [ ( ) [ } { [ ( ) ( ( ) [ ] ] { }
```

```
{}{[]}[](){([{}][]){}(){{}{}}[{()[]{}}]{}[[{()}]]()}()()[{}{}[(){}()][][](
){[{}[[]]()[()](){[]}()]((()))()({}){}(){([])[][]}(())[][()[]]()[]{}((([({{
}})]){[]})[[]]()[]()[()[]][]){()}(){}{()[[](()[{}[]])]({})[]{}}()()[][[{}()
]{(){()}[[]]{}[]}()]{}[]{}{}{}()(()()()){}}({}){}()(())[]
```

```
{}([][[][]])()([]()[{}](){()([({})()]())})(){}[]([][()[]]({}()(
())[]){(){}}{{}{}[][]}[][]{()[({}[])]{}}{{}[][]()[{()}[()][[()
{}()()]]{}][[]]([])[][{}](){}}{[]{()}{()}[[]()()[()[[]]](){}](
)}(){([])}{}{}){{}{()}()([])(()){}()}[[]{[]}]{()()}{}({{[]{}}(
[]{()})()}[][][[{}{}}]{}{([][]){[]{}}[]{}}[()][]{}({})[][[]
](()()([]))()({}()()[]{}[[]()(){{}[]()}[]{}{}[()]])([]{()}()){
}]{}{{}{{}}()([]{{{{}}}}[][][[]]][]()[()]{()(()){{()}}[()]()(
()){()}(){}[{}]{}{()[]{[]}{{()}}}()[([]){(()){}}[[()][][]]]({}
){}}{()})[]{}(()[](){{}{}(){[][]}}[[](){}])(){}}{{}[]{[{}]()[(
)]}[[[]{}]][{{}}[{}{}][][]{}]{}{}()[](()[])[]{}}{}([]{(){}}(()
[][[][][]()()()()[]()][[][()[]][{}]({}){}]))()()[[]({{}}[])]{(
())}({}{}{}{[]})[]{{}}()()({})[][()]{{}()[][]({}()()()){()()}(
)({}){}[[]][]}{{{}}({{}}[]({}{}{}{[]})[]){()[]()}({}[])(){}[]}
){}{}[](()()[[[][][]]{}{(({}))){}}(()){}()({()(){}()(())})][][]
{}[][]{}{{}[()[]{}][(()())[[{}([])]]{[]{}}({})]([][[]]{()})[{}
]{{}{}}}()[()][]()[]{}{})({}[{}]()[]){}{}({}{()}){()({})}[]{{{
}}}{{}}{{}}{}()[{()(){(())[]}{}}()[[[]]]{}[](()[]){{}{}}(){}[
[]]())[{[]{}[][]{{}()}[]}({}[])]([{{{}()}}]()()[({})]{}()){{}}
([]([])[][]{{}({[]{{}[]{}(({}))}}))}({}{([])(())[{}[]][[]]([]{(
)}[])}(([]))[])([[()]]{[]{}}(()({})[]()[[]]))){}(({{}()[][()[]
]}())(([])()())(){}{()}{}{}(([])){}[])({}){}(()()())(){}({}[][
](())(){{}{[[]]}(){}[])((()()[][()[()]{}()]{{}}){})([{}]{}({}
()[[[]()]]{[]}{})(){}()[]){[]}}[][][()]{[]()[]{}([]){}([]){[[[
]]]}}}][]{}()()()[{}]({})(){}{{[]}([](){{(){}}[][{}]}[])()()()
[()({}())(){()(())){}[[][]]}[]{[()]}[]{{()}}(){}(()){{}}[[]()]]
[]()()(){[]()([{}])}{}{}{}[{}][({})]{}{[]{}()}(){}[](){}{}{}{(
{{(()[[]]{}())()}(){(){}}{}})[]{{}([]{})[(){}{}()]}{[]}[{}][]}
}[([]())()[][]{}[[]]{}[]][]{[(){}](({}))(){}}([[]()]()(){})({}
)()[{}()][{}[]([]())](){}(){{{}}}({[]{}[]}()[()([])][({}{{}}{[
][]}{[]{}[{}[]]{[]}()}())()()()(){{}}][[(()[{}]){}][]{}]{}{{}[
[]]{}[][]}{}{})[[](([])[])]{}[](){}()[[]{}[](){()}({}{()}[][](
)(()[{}[{}]()(){}())[[]()][]){(){}[]()[]}{{}}]({}[]{}(){([])}({
}{{}(){}}()))){}
```



aaa{aaa}aaa[aaa]aaa[aaa{aaa}aaa(aaa(aaa)abaa)a’aa]aa;a[a`aa[a!aa[#aaa]aa@a(aASDaa)aaa]aaa]aaa