Nested and 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++, these delimiters are used in matching pairs, so (…), […] or {…}, and never in the form)…(,]…[or }…{, or otherwise mixed; for example, (3, 5]. 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 }. You will never find an opening delimiter without a matching closing delimiter, 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:

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

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

1. There are as many opening delimiters of each style as there are closing delimiters.
2. There does not appear to be a *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 examples such as:

}])	A closing delimiter without a matching opening opening.
[} [) (] (} {] {]	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.

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

Determine which of the following have correctly nested matching delimiters:

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

Angled brackets and context sensitivity

You will have noticed that under certain circumstances, angled brackets (< and >) are also used as delimiters; however, angled brackets may also be used as less-than or greater-than operators, respectively, or with the left bit-shift and right bit-shift operators (<< and >>), respectively. These two symbols are referred to as angled brackets only if they are being used for grouping. Three examples where angled brackets are used for grouping in C++ are the following:

```
#include <iostream>
template <typename T> T function_name( T x );
std::set<int> variable_name{};
```

You will likely only be familiar with the first case, as the second and third involve the use of templates, another feature in C++.

For the six standard delimiters, there are no cases where they are not used for grouping statements and expressions, and must therefore always satisfy the properties described above. For the < and > characters, their interpretation is said to be *context sensitive*, meaning how they are interpreted depends on how they are used. This is the same with the spoken English language. If an individual says “the sky is red” versus “we read the book”, the context indicates how to interpret the pronounced word. Consider, however, the example given by Wayne Leman: “Visiting relatives can be exhausting.” Here, there is insufficient context to disambiguate the word “visiting” as it could be a verb “Visiting one’s relatives can be exhausting.” or an adjective “Relatives who are visiting can be exhausting.” Of course, there may be additional context outside the sentence in question: if you are coming back from an airport after a vacation (presumably after visiting your relatives), you probably meant the first, while if you are coming back from an airport after dropping someone off (presumably your relatives), you probably meant the second. Programming languages, however, cannot have such ambiguities. There are very strict rules for when < means a less-than operator, << means a left bit-shift operator, and when < means an opening angled bracket delimiter and << means two opening angled bracket delimiters. That will be for another course, however.

But why? Is the compiler so dumb?

You may write the program

```
#include <iostream>

int main();

int main() {
    std::cout << "Hello world!" << std::endl;

    return 0;
```

and wonder why it does not compile. Obviously you “meant” for the function body to end after the statement `return 0`. After all, what else could it do? The point is, a programming language is not written for trivial projects, but rather for large complex projects used by companies and academics to perform tasks like running a nuclear reactor, flying an aircraft or detecting signs of cancer in a scan. If the compiler is allowed to make guesses as to when, for example, a closing parenthesis, bracket or brace should appear, such a guess may be incorrect, and could change the entire sense of the program. Here is one example where the wrong rule may put the closing delimiter in the wrong location. Suppose that the rule is, at the end of the program, if there are missing closing delimiters, add as many is necessary. Where should the missing closing brace in this example go, and does the rule put that brace in that location?

```
bool sort_three( double &x, double &y, double &z );

bool sort_three( double &x, double &y, double &z ) {
    double t{};
    // Were the three originally sorted?
    bool was_sorted{true};

    // If x > y, swap x and y
    if ( x > y ) {
        t = x;
        x = y;
        y = t;
        was_sorted = false;
    }

    // If y > z, swap y and z
    if ( y > z ) {
        t = y;
        y = z;
        z = t;
        was_sorted = false;

    // If x > y, swap x and y
    if ( x > y ) {
        t = x;
        x = y;
        y = t;
    }

    return was_sorted;
}
```

The correct location for the closing brace is where indicated below in red, but if `was_sorted` was not present and the function returned `void`, the closing brace could go at the end.

```
bool sort_three( double &x, double &y, double &z );

bool sort_three( double &x, double &y, double &z ) {
    double t{};
    // Were the three originally sorted?
    bool was_sorted{true};

    // If x > y, swap x and y
    if ( x > y ) {
        t = x;
        x = y;
        y = t;
        was_sorted = false;
    }

    // If y > z, swap y and z
    if ( y > z ) {
        t = y;
        y = z;
        z = t;
        was_sorted = false;

        // If x > y, swap x and y
        if ( x > y ) {
            t = x;
            x = y;
            y = t;
        }
    }

    return was_sorted;
}
```

If the rule was just to add an extra brace at the end, this would result in a bug. Relying on such questionable behavior may be even more dangerous.

Algorithm for determining if delimiters are nested and matching

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.

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.

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

(	{	[	(	{	(														

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.

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

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.

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

[	{																		