How does the Calculator Compiler work?

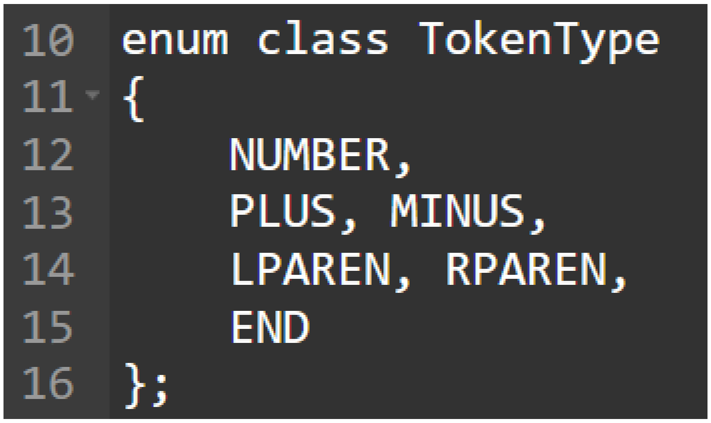
First thing to note, I used what’s called Top-Down. This video does a great job explaining it (Skip to 5:00) https://www.google.com/search?client=firefox-b-1-d&sca_esv=577472328&sxsrf=AM9HkKn7vH1sLctZmnFE2y1srTD31eiLXw:1698514009103&q=recursive+top+down+parser&tbm=vid&source=lnms&sa=X&ved=2ahUKEwjsjKLhoZmCAxXIJUQIHanuA1AQ0pQJegQICxAB&biw=2048&bih=999&dpr=1.25#fpstate=ive&vld=cid:f9a25df1,vid:iddRD8tJi44,st:0

Classes, Struct and Enums

Here are the fundamentals I used and what each of them mean

This is the “marker” which identifies which token we are looking at and nothing more.

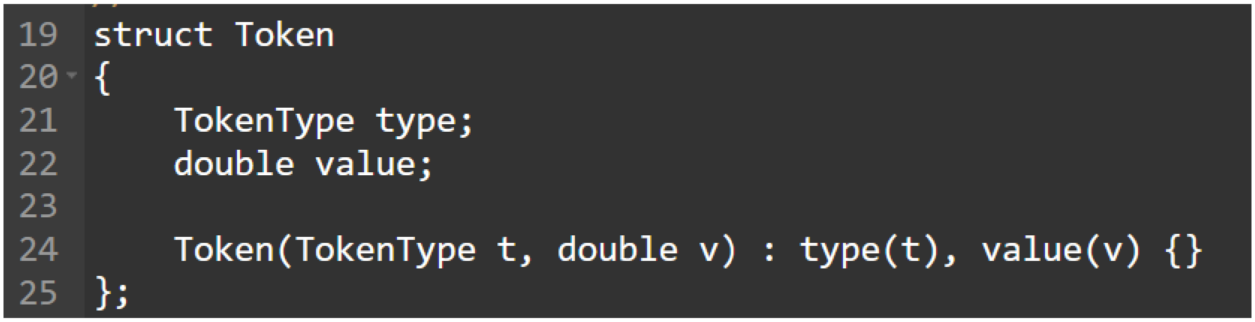
Note: END means we reach the end of the expression and terminate.



```
enum class TokenType
{
    NUMBER,
    PLUS, MINUS,
    LPAREN, RPAREN,
    END
};
```

This is the actual token. Each token is hashed with a value, but for this simplified code, the value is only used for integers and its value.

Since this is the actual token, it holds the enum "marker" as well.

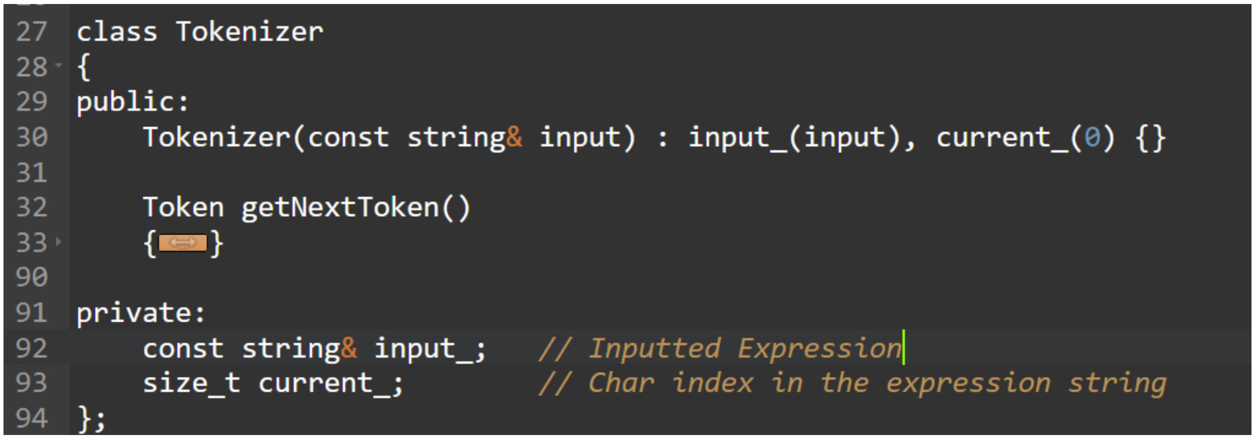


```
struct Token
{
    TokenType type;
    double value;

    Token(TokenType t, double v) : type(t), value(v) {}
};
```

The Tokenizer takes in the equation that we inputted and scrolls through it's index one by one in the "getNextToken()"

The getNextToken() method looks at the current index of the string, figures out what it is, and responds by returning the "marker" saying what the token type is AND inputs the integers in the token.

This means the Tokenizer: Scrolls through the string, sees what it is, returns the token (which is the Token struct that holds the "marker" and int value)



```
class Tokenizer
{
public:
    Tokenizer(const string& input) : input_(input), current_(0) {}

    Token getNextToken()
    {⟺}

private:
    const string& input_;    // Inputted Expression
    size_t current_;         // Char index in the expression string
};
```

The parser is what runs the program (for this code example only)

Parser class takes in the Tokenizer class as an object reference (Remember, the tokenizer has our inputted equation AND is what returns what the token we are looking at actually is) By default, we assume that we are at the “end”.

Parse() is where the execution of the CTG happens

Expression, term, factor is the following CTG which is (for this code example only)

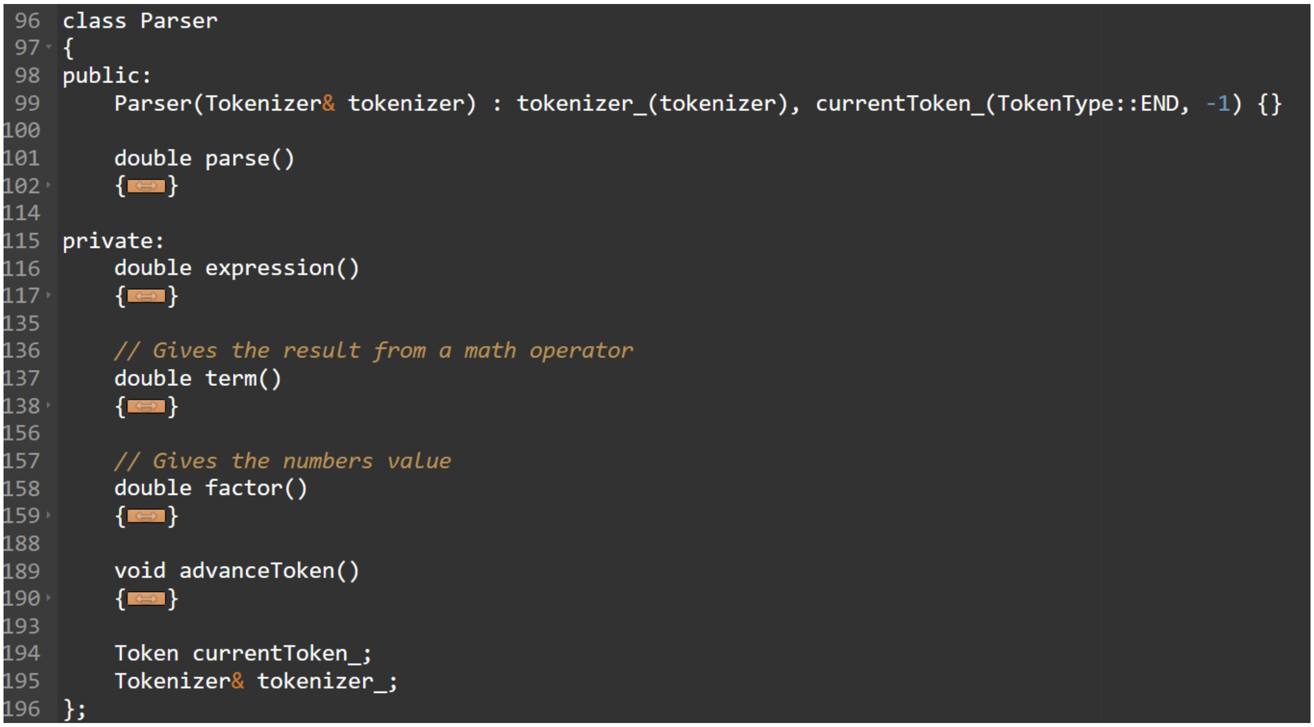
expression ::= term {‘+’ | ‘-‘ } term

term ::= factor | {‘+’ | ‘-‘ } factor

factor ::= Number | "(" expression* "")"

AdvanceToken() calls the tokenizers “getNextToken()” member

Tokenizer_ is the object I made to hold the tokenizer that is passed in through the parser



```
class Parser
{
public:
    Parser(Tokenizer& tokenizer) : tokenizer_(tokenizer), currentToken_(TokenType::END, -1) {}

    double parse()
    {↔}

private:
    double expression()
    {↔}

    // Gives the result from a math operator
    double term()
    {↔}

    // Gives the numbers value
    double factor()
    {↔}

    void advanceToken()
    {↔}

    Token currentToken_;
    Tokenizer& tokenizer_;
};
```

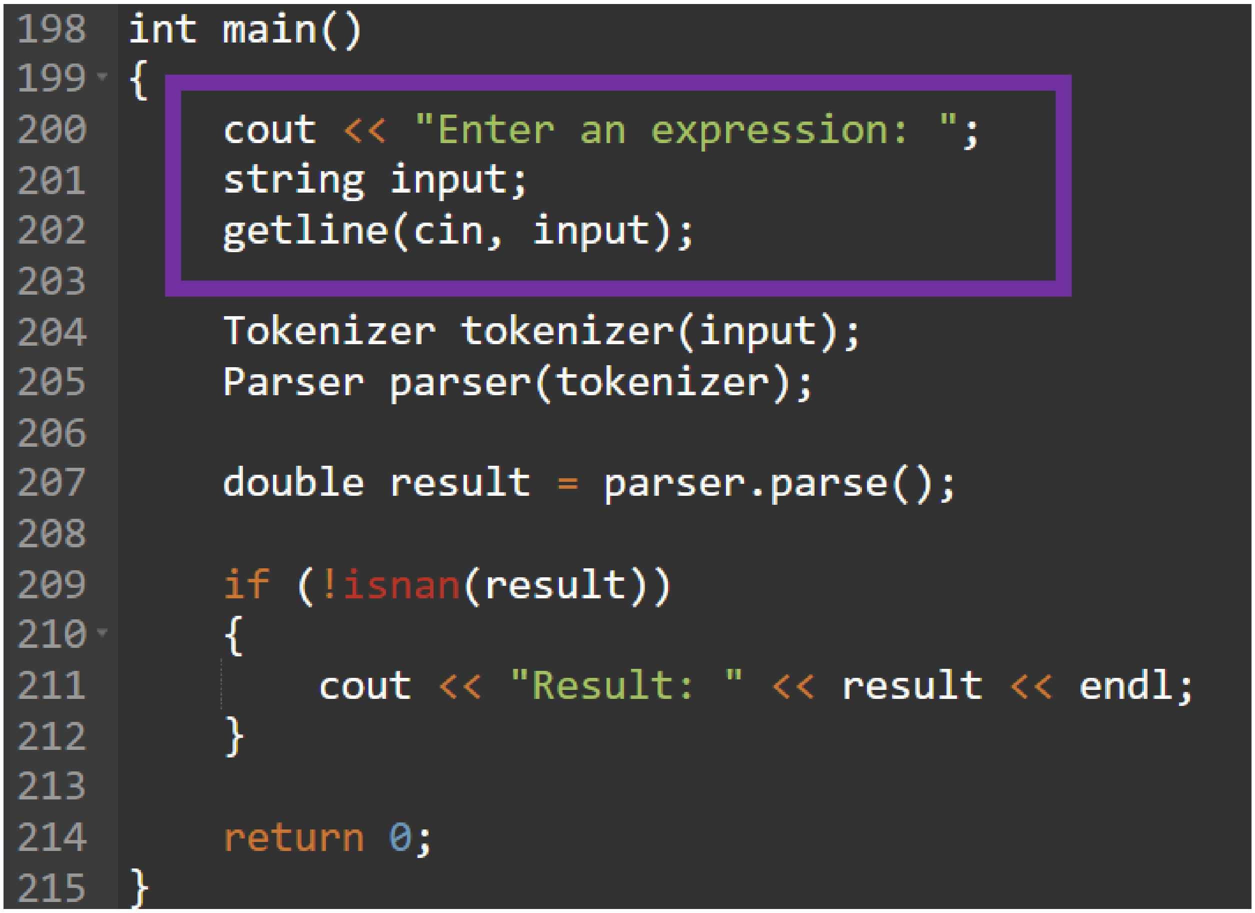
How the hell does it actually work

If you’re looking at some of this still confused, don’t worry. That was just letting you know what each class has/does.

To explain how it works, we are going to start in main and go line-by-line jumping around each line to show every detail on what’s happening. Do note that I included code line numbers for simplicity and will say things like “left off at line ---, and jump to line ---“.

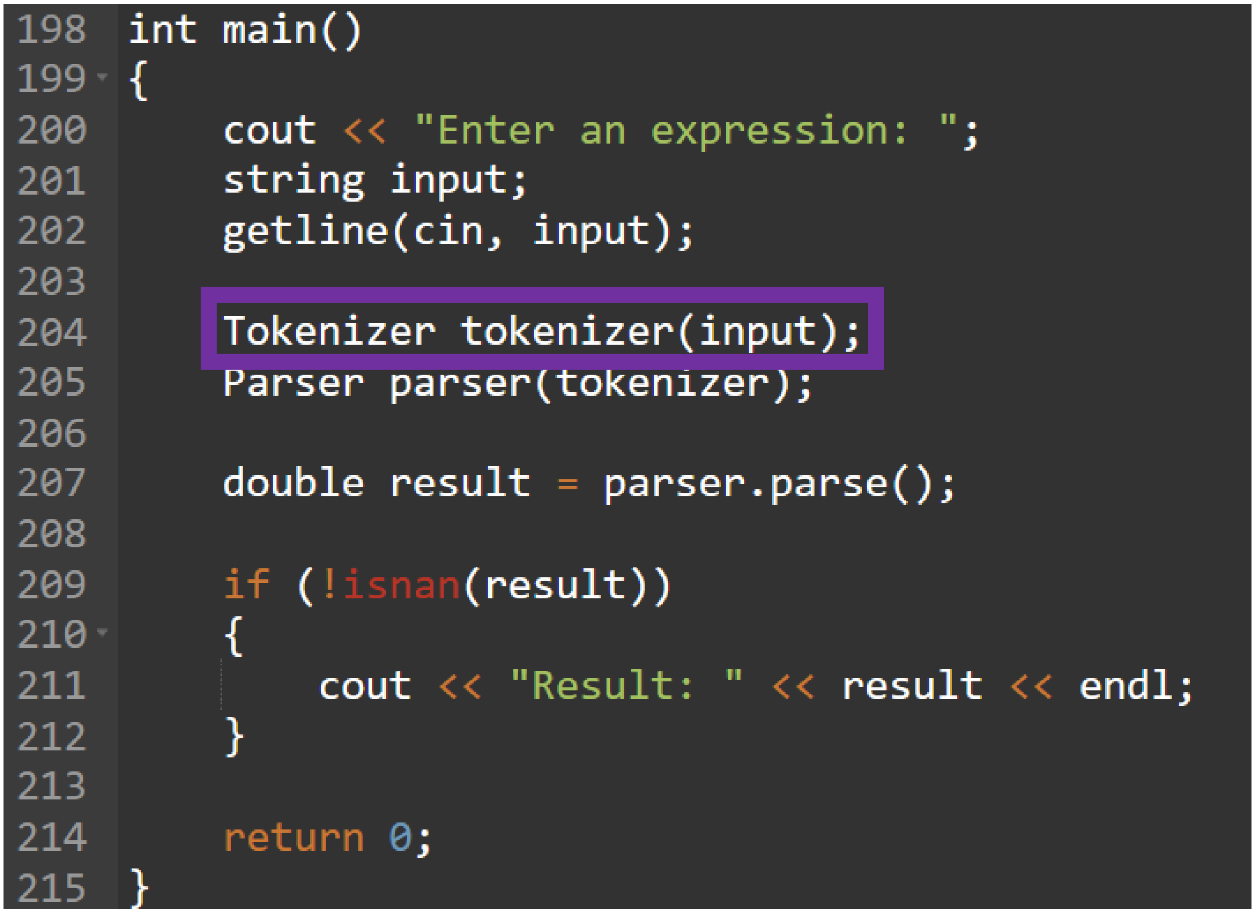
In this example, imagine the input as any number plus any number.

In main, we start with the simple input that we all know how to do.



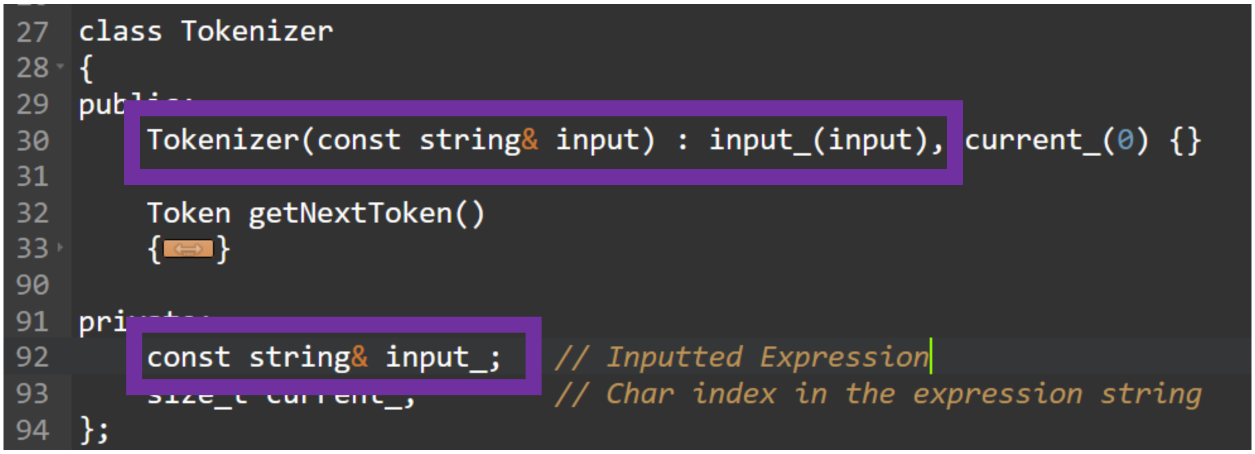
```
198 int main()
199 {
200     cout << "Enter an expression: ";
201     string input;
202     getline(cin, input);
203 
204     Tokenizer tokenizer(input);
205     Parser parser(tokenizer);
206 
207     double result = parser.parse();
208 
209     if (!isnan(result))
210     {
211         cout << "Result: " << result << endl;
212     }
213 
214     return 0;
215 }
```

Our input is then passed into the tokenizer object. Let’s see what happens when we do that by jumping to the tokenizer class



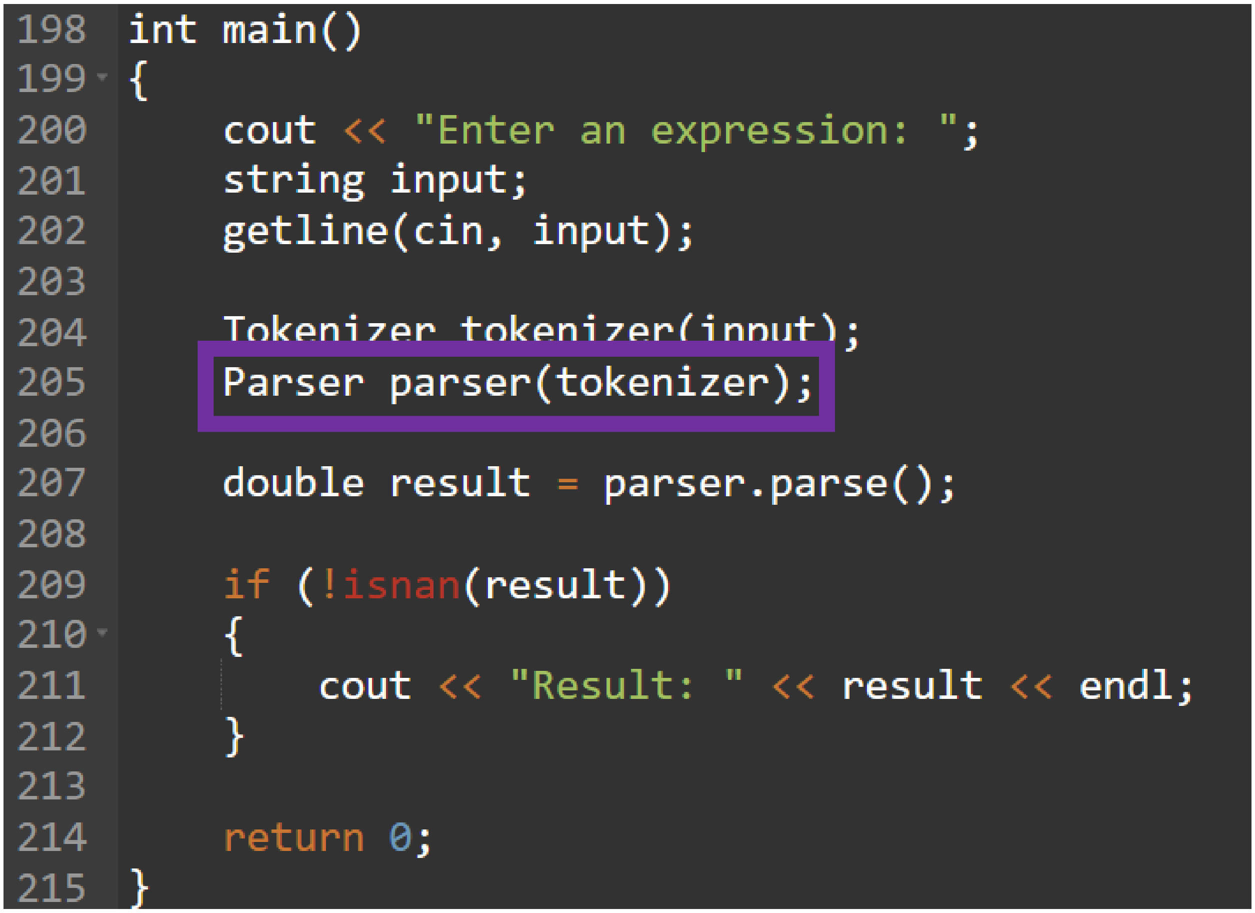
```
int main()
{
    cout << "Enter an expression: ";
    string input;
    getline(cin, input);

    Tokenizer tokenizer(input);
    Parser parser(tokenizer);

    double result = parser.parse();

    if (!isnan(result))
    {
        cout << "Result: " << result << endl;
    }

    return 0;
}
```

The string is passed in and now “input_” has our expression



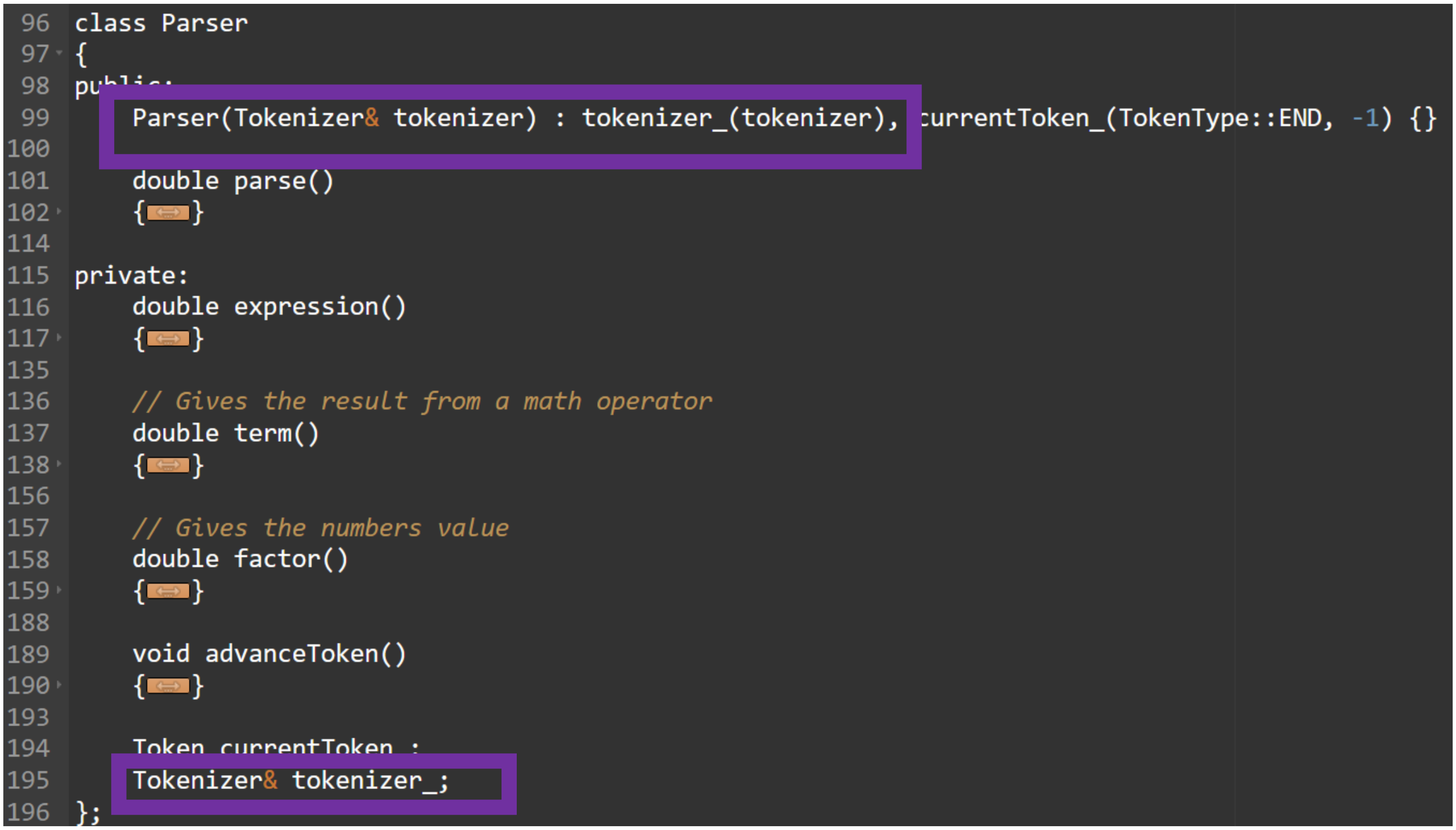
```
class Tokenizer
{
public:
    Tokenizer(const string& input) : input_(input), current_(0) {}

    Token getNextToken()
    {⟺}

private:
    const string& input_;   // Inputted Expression
    size_t current_;        // Char index in the expression string
};
```

Now we advance to the next line of code which is passing the tokenizer class to the parser. Lets jump to the parser class to see what happens



```
int main()
{
    cout << "Enter an expression: ";
    string input;
    getline(cin, input);

    Tokenizer tokenizer(input);
    Parser parser(tokenizer);

    double result = parser.parse();

    if (!isnan(result))
    {
        cout << "Result: " << result << endl;
    }

    return 0;
}
```

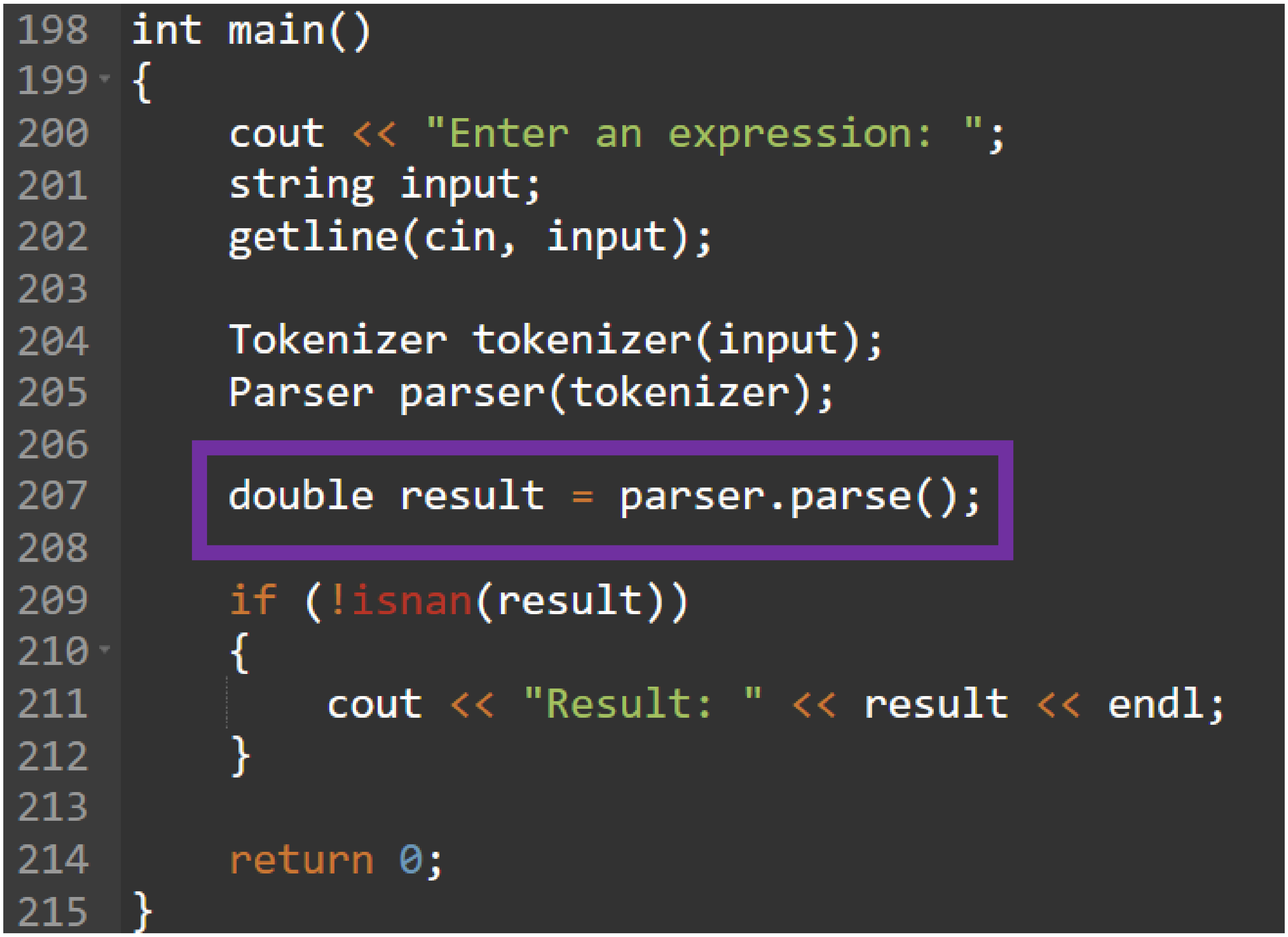
When we pass in the tokenizer class (remember, this has out equation that we inputted) The tokenizer object in parser is initialized to the tokenizer class we passed



```
class Parser
{
public:
    Parser(Tokenizer& tokenizer) : tokenizer_(tokenizer), currentToken_(TokenType::END, -1) {}

    double parse()
    {⇔}

private:
    double expression()
    {⇔}

    // Gives the result from a math operator
    double term()
    {⇔}

    // Gives the numbers value
    double factor()
    {⇔}

    void advanceToken()
    {⇔}

    Token currentToken_;
    Tokenizer& tokenizer_;
};
```

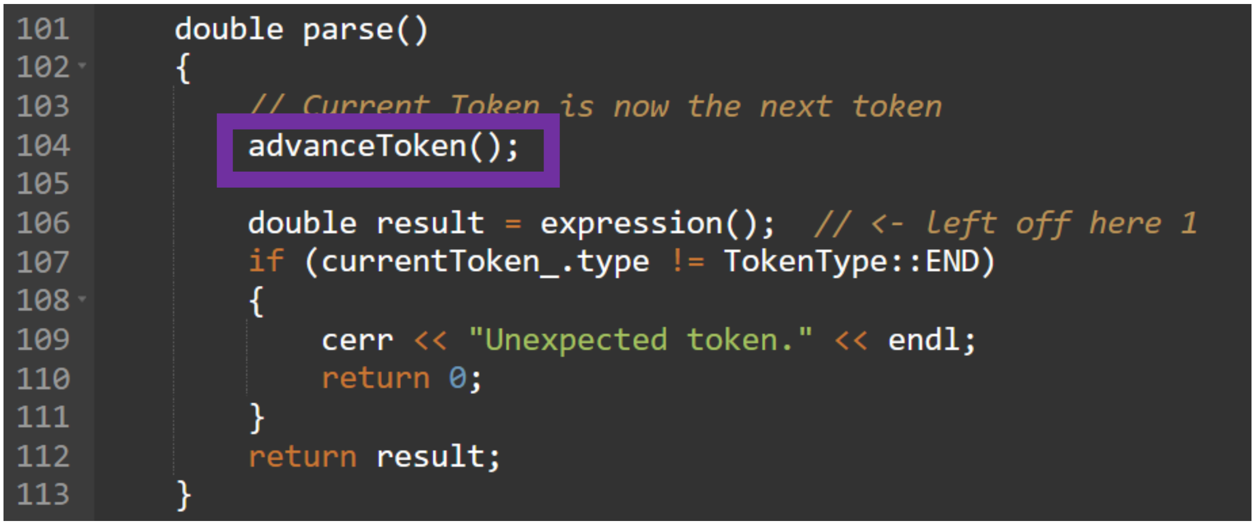
Note, no algorithmic things have happened yet. All we have done so far is get input and send it around to everywhere its needed. Now we use the parse() function which is where everything truly “starts”

Lets jump back to parser to see what happens



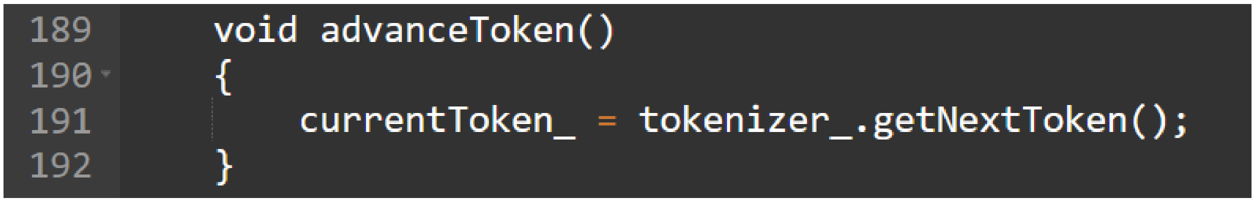
```cpp
int main()
{
    cout << "Enter an expression: ";
    string input;
    getline(cin, input);

    Tokenizer tokenizer(input);
    Parser parser(tokenizer);

    double result = parser.parse();

    if (!isnan(result))
    {
        cout << "Result: " << result << endl;
    }

    return 0;
}
```

Inside of our parse() function the first thing we do is use the function advanceToken()



```
    double parse()
    {
        // Current Token is now the next token
        advanceToken();

        double result = expression();  // <- left off here 1
        if (currentToken_.type != TokenType::END)
        {
            cerr << "Unexpected token." << endl;
            return 0;
        }
        return result;
    }
```

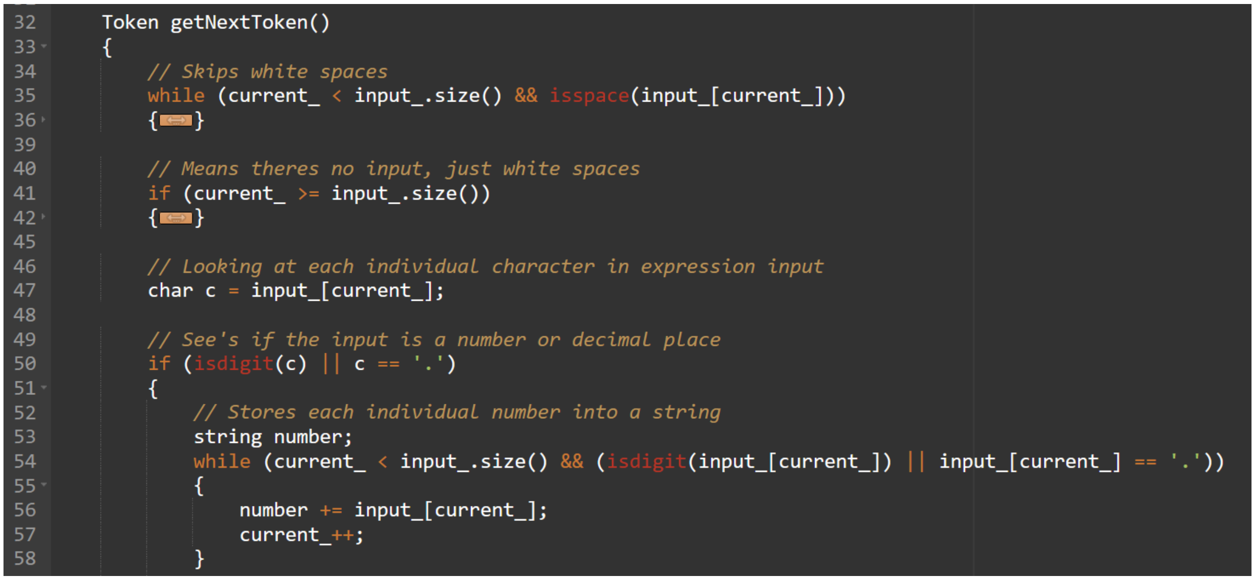
advanceToken() is a member of parser that uses the tokenizers member, getNextToken(). Lets jump to the tokenizer class to see what this does (line 32)



```
    void advanceToken()
    {
        currentToken_ = tokenizer_.getNextToken();
    }
```

The closed down code is syntax to skip whitespaces, don't worry about it

Remember, our input in this example is a number plus a number. So our first token should be a number

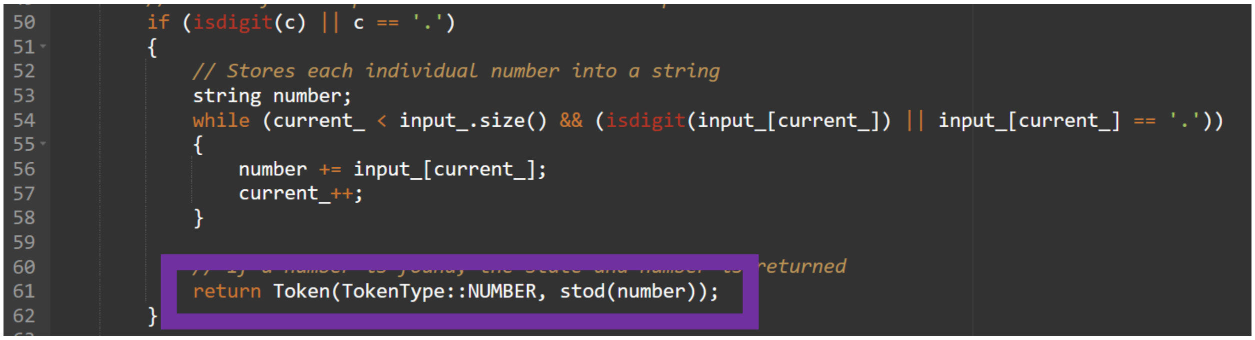


```
    Token getNextToken()
    {
        // Skips white spaces
        while (current_ < input_.size() && isspace(input_[current_]))
        {↔}

        // Means theres no input, just white spaces
        if (current_ >= input_.size())
        {↔}

        // Looking at each individual character in expression input
        char c = input_[current_];

        // See's if the input is a number or decimal place
        if (isdigit(c) || c == '.')
        {
            // Stores each individual number into a string
            string number;
            while (current_ < input_.size() && (isdigit(input_[current_]) || input_[current_] == '.'))
            {
                number += input_[current_];
                current_++;
            }
```

The tokenizer takes the index we are looking at (right now starting at 0) and checks if that index is a number.

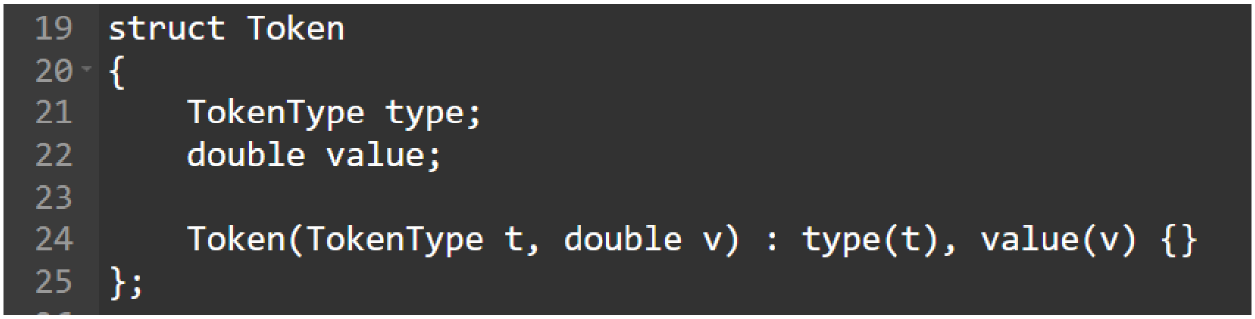
Notice how current_ gets added by one before we pass in the results we find. This is so that when we call getNextToken() again, it start at the index after the one we just identified.

In this case it is so what is does is returns a Token struct object that holds the token "marker" (enum) and the integers value (just imagine any number you want).



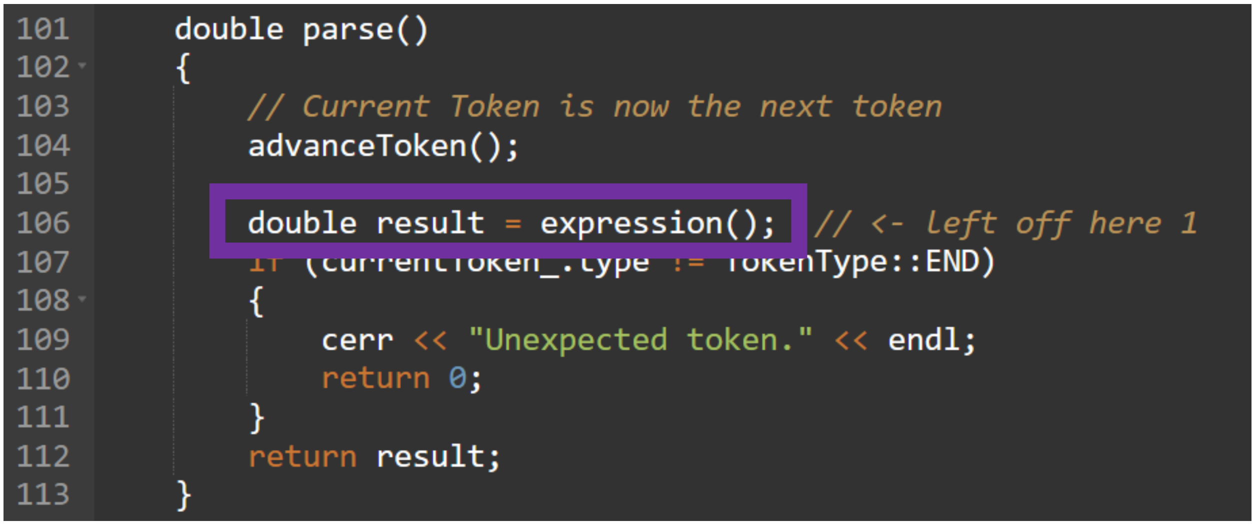
```
        if (isdigit(c) || c == '.')
        {
            // Stores each individual number into a string
            string number;
            while (current_ < input_.size() && (isdigit(input_[current_]) || input_[current_] == '.'))
            {
                number += input_[current_];
                current_++;
            }

            // If a number is found, the state and number is returned
            return Token(TokenType::NUMBER, stod(number));
        }
```

Remember, Token is a struct that holds the identifier of the token and its integer value



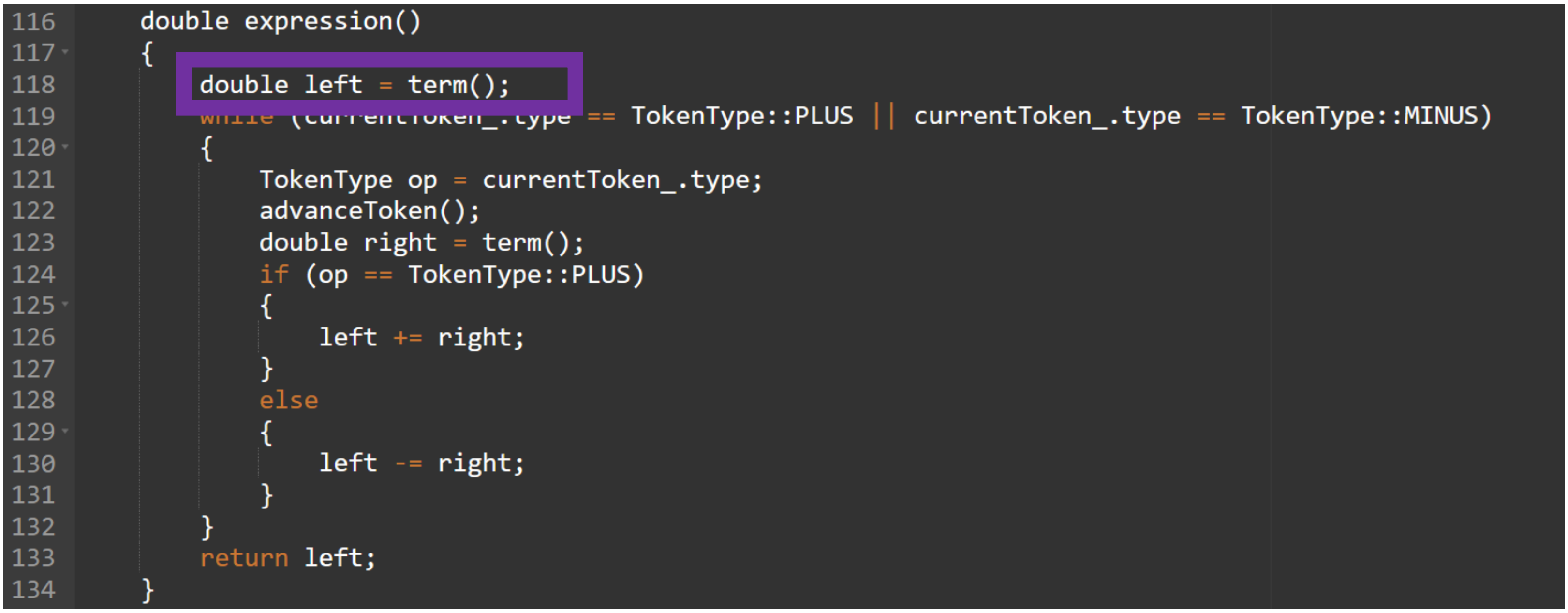
```
struct Token
{
    TokenType type;
    double value;

    Token(TokenType t, double v) : type(t), value(v) {}
};
```

Now that we returned a Token struct which holds the identifier and the value, we are back here at the parse() function. We now call the parsers member expression()



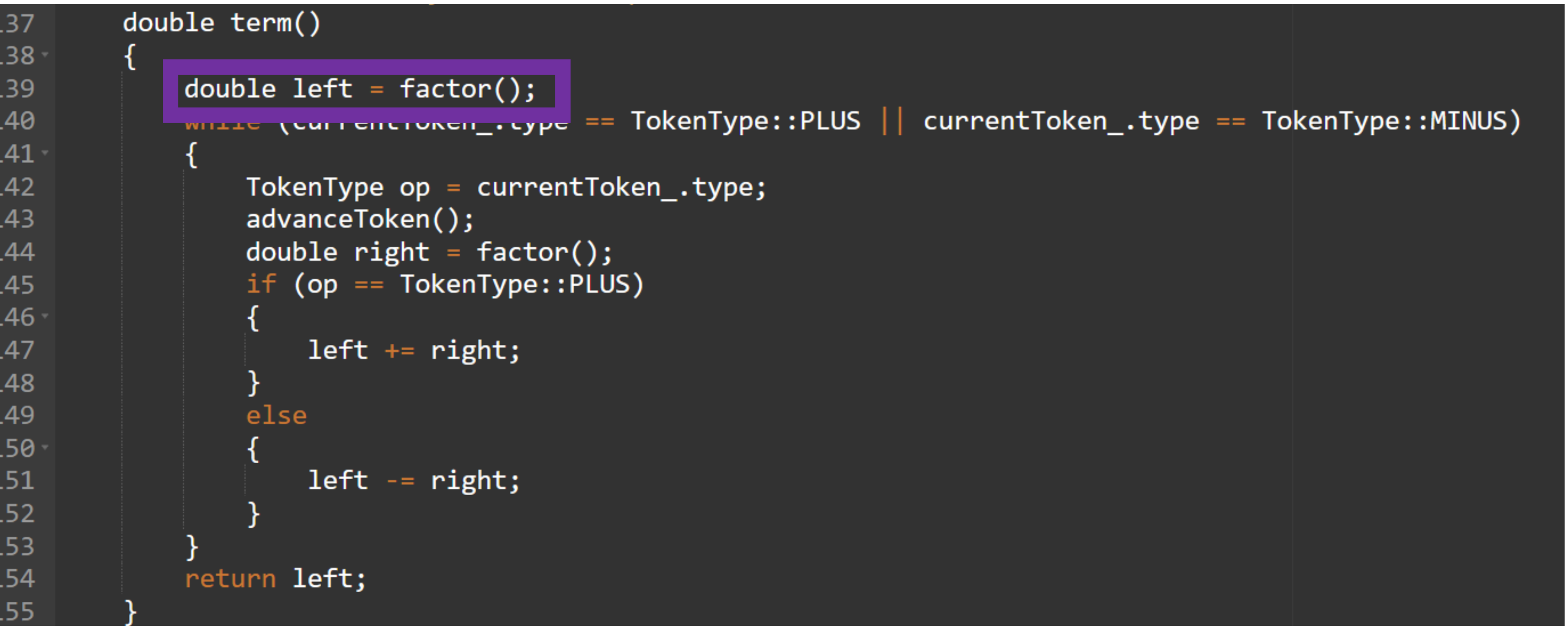
```
    double parse()
    {
        // Current Token is now the next token
        advanceToken();

        double result = expression(); // <- left off here 1
        if (currentToken_.type != TokenType::END)
        {
            cerr << "Unexpected token." << endl;
            return 0;
        }
        return result;
    }
```

Inside expression we **first** call term



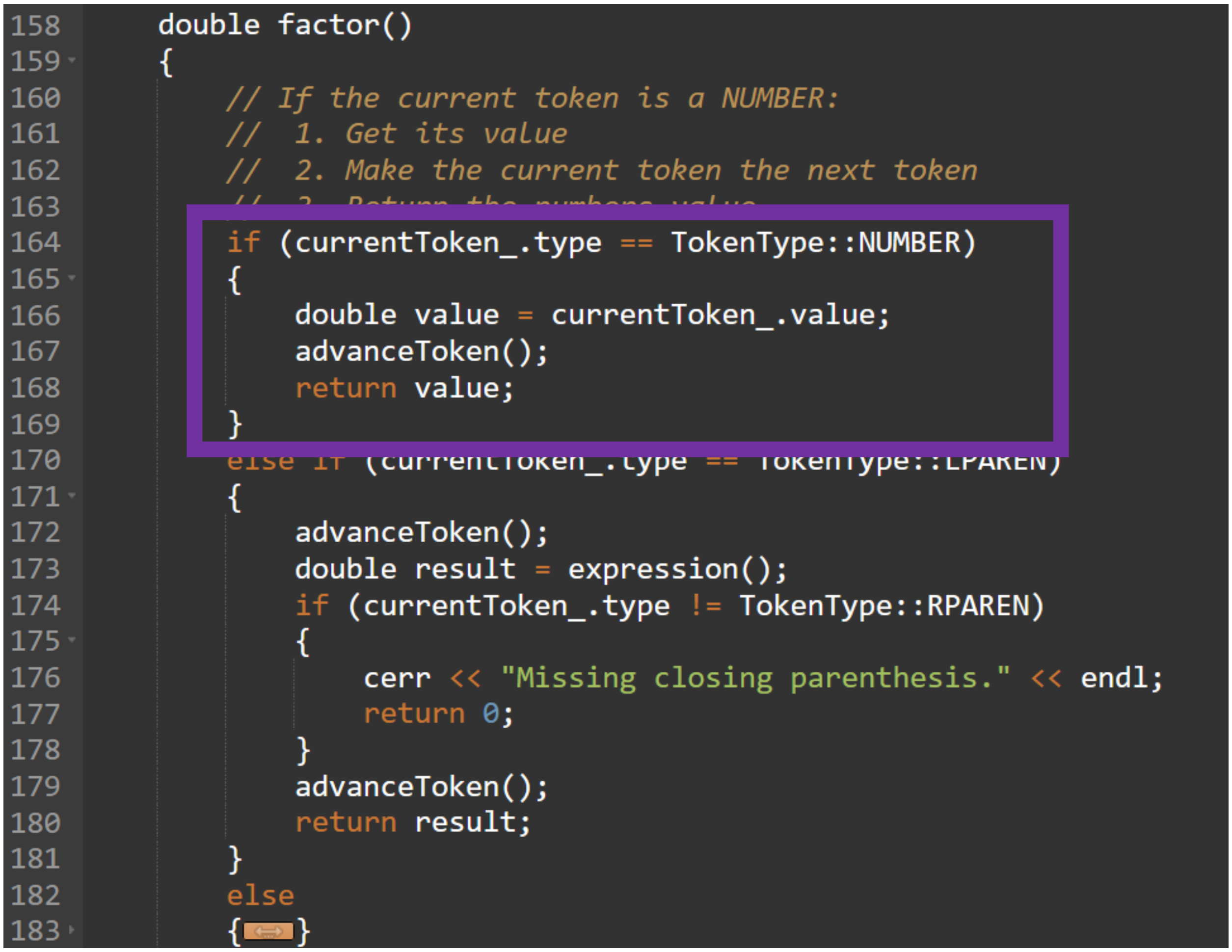
```
    double expression()
    {
        double left = term();
        while (currentToken_.type == TokenType::PLUS || currentToken_.type == TokenType::MINUS)
        {
            TokenType op = currentToken_.type;
            advanceToken();
            double right = term();
            if (op == TokenType::PLUS)
            {
                left += right;
            }
            else
            {
                left -= right;
            }
        }
        return left;
    }
```

Inside of term we **second** call factor()



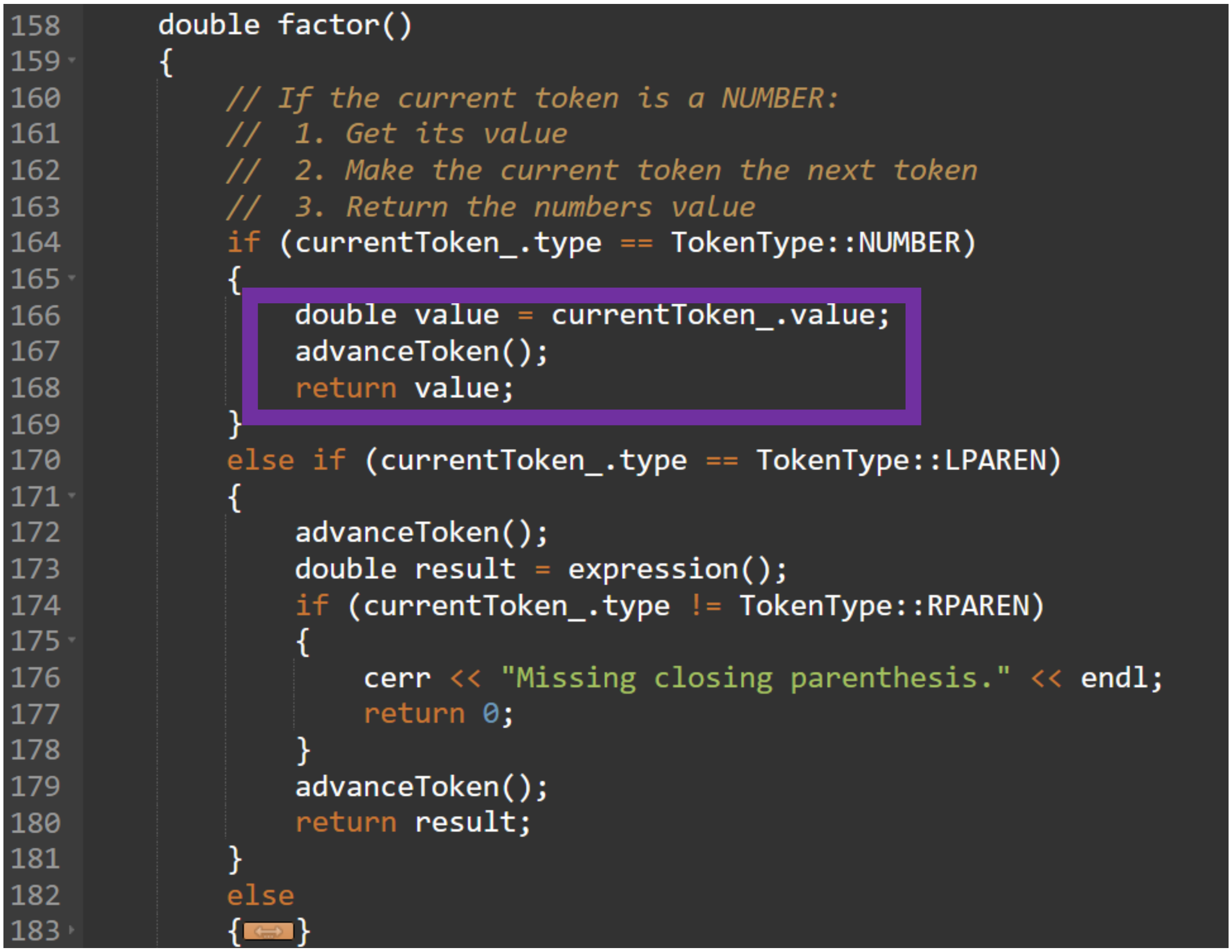
```
    double term()
    {
        double left = factor();
        while (currentToken_.type == TokenType::PLUS || currentToken_.type == TokenType::MINUS)
        {
            TokenType op = currentToken_.type;
            advanceToken();
            double right = factor();
            if (op == TokenType::PLUS)
            {
                left += right;
            }
            else
            {
                left -= right;
            }
        }
        return left;
    }
```

Lastly we are now in factor. Remember, our demo input here is a number plus a number. Our currentToken_ is the struct Token that is holding our enum identifier (type). So we know it's a number and look at the first if statement.



```
    double factor()
    {
        // If the current token is a NUMBER:
        //  1. Get its value
        //  2. Make the current token the next token
        //  3. Return the numbers value
        if (currentToken_.type == TokenType::NUMBER)
        {
            double value = currentToken_.value;
            advanceToken();
            return value;
        }
        else if (currentToken_.type == TokenType::LPAREN)
        {
            advanceToken();
            double result = expression();
            if (currentToken_.type != TokenType::RPAREN)
            {
                cerr << "Missing closing parenthesis." << endl;
                return 0;
            }
            advanceToken();
            return result;
        }
        else
        {↔}
```

Next we get the integers value that was passed in and advance to the next token and return the number.



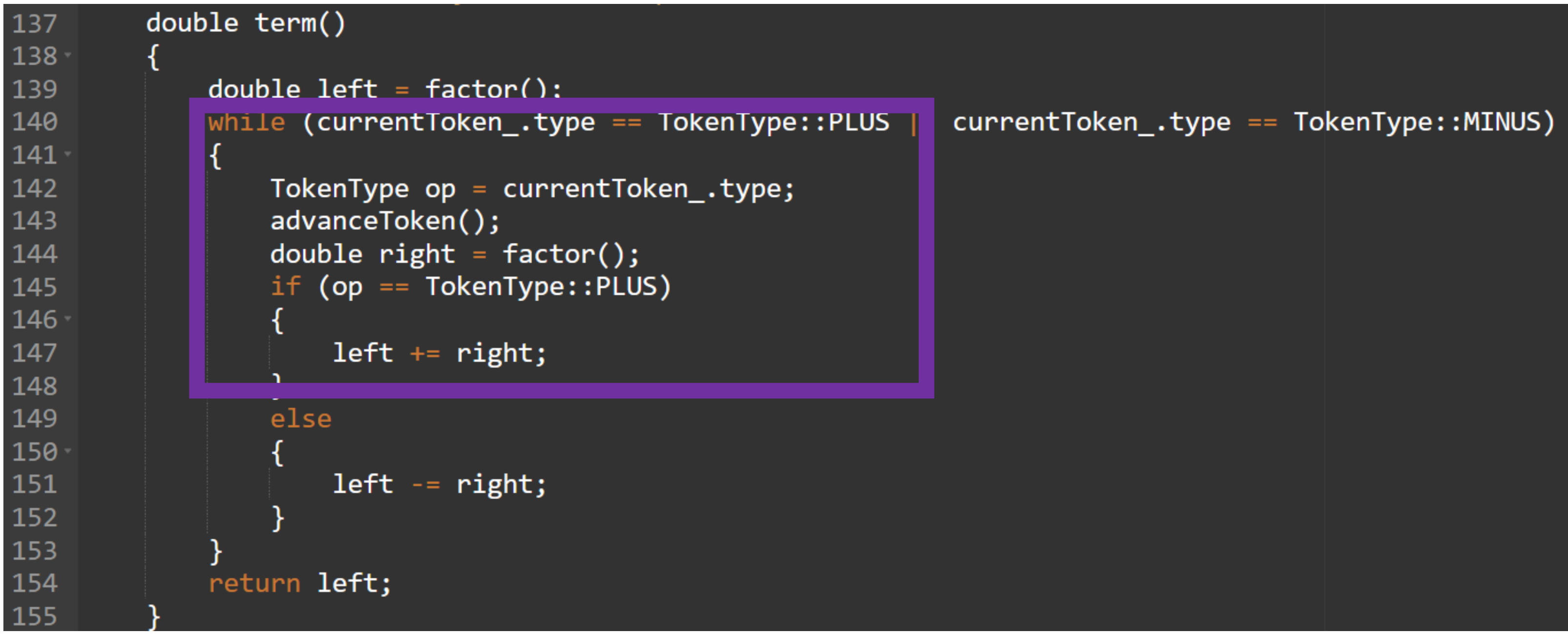
```
    double factor()
    {
        // If the current token is a NUMBER:
        //  1. Get its value
        //  2. Make the current token the next token
        //  3. Return the numbers value
        if (currentToken_.type == TokenType::NUMBER)
        {
            double value = currentToken_.value;
            advanceToken();
            return value;
        }
        else if (currentToken_.type == TokenType::LPAREN)
        {
            advanceToken();
            double result = expression();
            if (currentToken_.type != TokenType::RPAREN)
            {
                cerr << "Missing closing parenthesis." << endl;
                return 0;
            }
            advanceToken();
            return result;
        }
        else
        {↔}
```

The number gets returned to factor and since we have no plus or minus operations going on yet, we return what received.

HOWEVER, remember how we called advanceToken() previously? Well now we are now looking at our NEXT token, not our current token. Remember what that does, it observes the string we inputted in main and goes through it one by one via string index.

We have our current token taken care of. It was a number. Done. Don’t need it again. Left holds the numbers value.

In this example, we are doing a number plus a number so now our token is a PLUS. We go to the loop that does what plus does AND AGAIN advance to the next token BECAUSE we want to see what to do with the token we just got (plus sign) and the previous token we had (a number) So we call factor again, which will do the exact same thing as listed above and give us a number

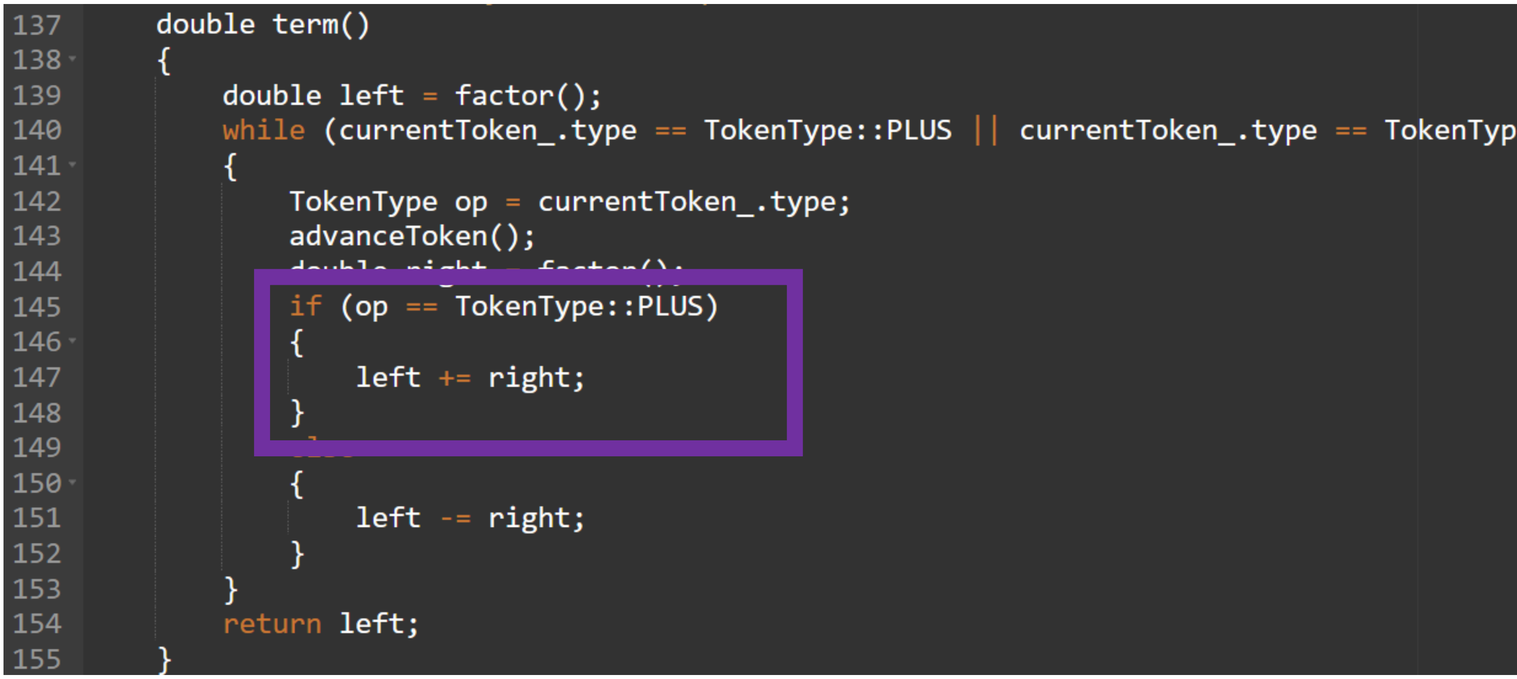


```
double term()
{
    double left = factor();
    while (currentToken_.type == TokenType::PLUS || currentToken_.type == TokenType::MINUS)
    {
        TokenType op = currentToken_.type;
        advanceToken();
        double right = factor();
        if (op == TokenType::PLUS)
        {
            left += right;
        }
        else
        {
            left -= right;
        }
    }
    return left;
}
```

op is what holds the previous token which in this code is set up to be an operator.

Right calls factor which sends us another number (we have a number + a number in our expression. We just evaluated the first number and the plus sign, now we get returned the last number).

After we have both the left number, operator, and right number, we do the if statement and return the value

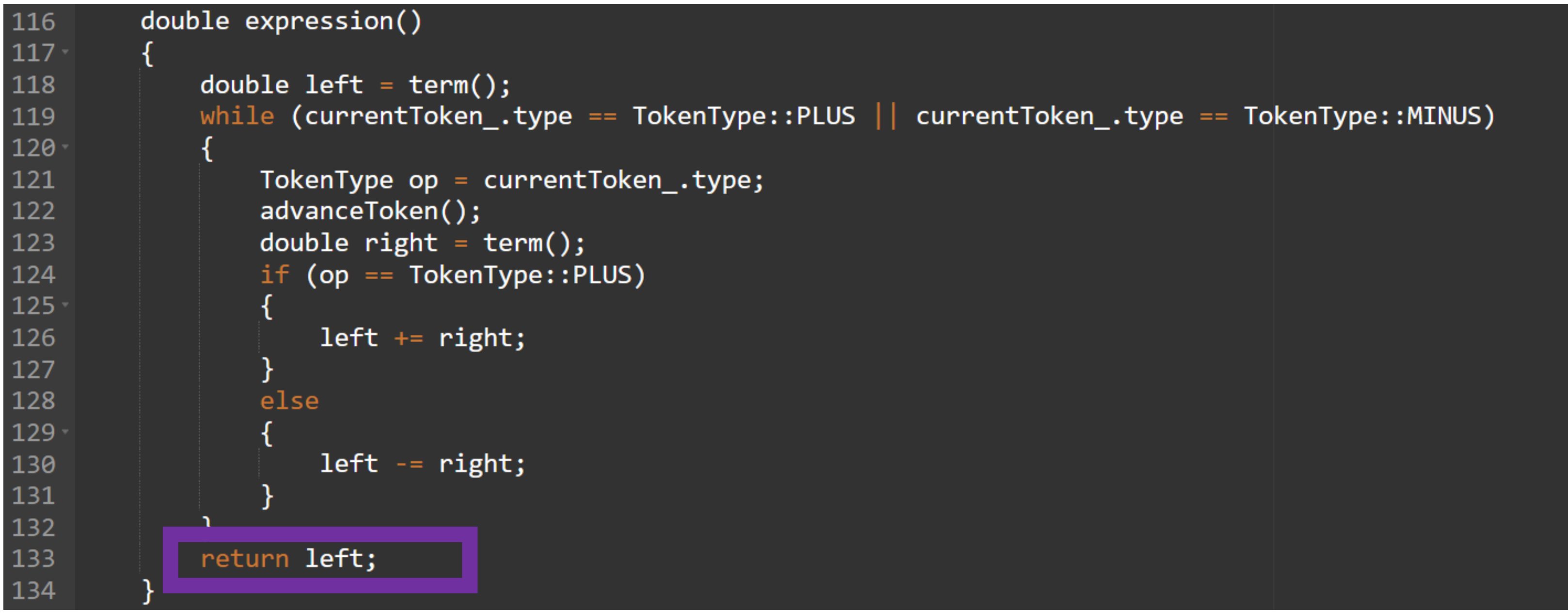


```
    double term()
    {
        double left = factor();
        while (currentToken_.type == TokenType::PLUS || currentToken_.type == TokenTyp
        {
            TokenType op = currentToken_.type;
            advanceToken();
            double right = factor();
            if (op == TokenType::PLUS)
            {
                left += right;
            }
            else
            {
                left -= right;
            }
        }
        return left;
    }
```

The value of the operation is returned to expression().

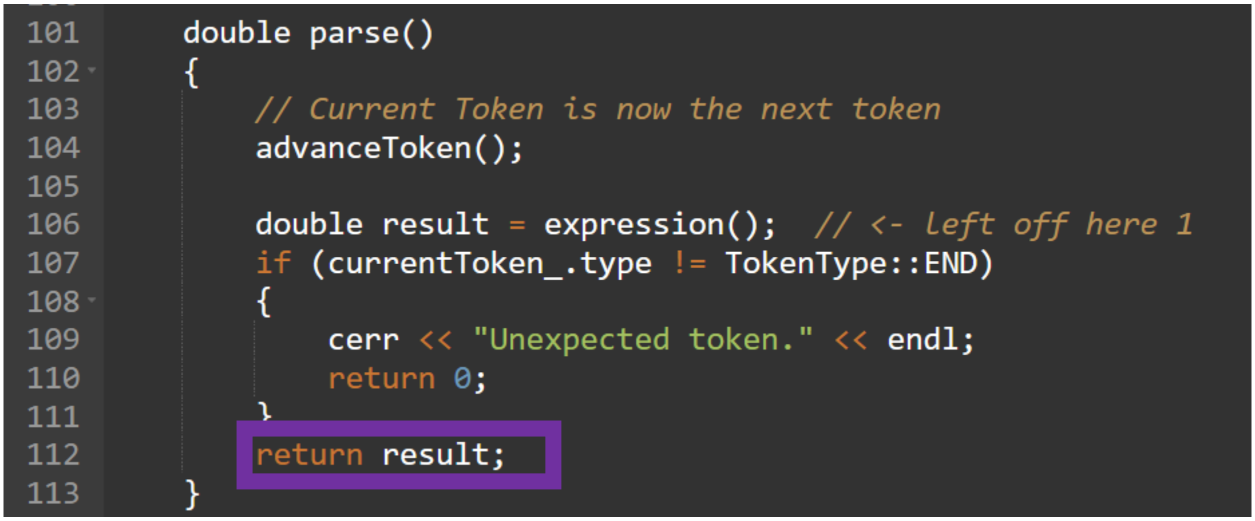
(If you are wondering what the rest of the code is in expression, that will take too long to explain over typing it up, so ill explain that to you guys later)

Expression returns the RESULT of what we just processed (which was a number plus a number)



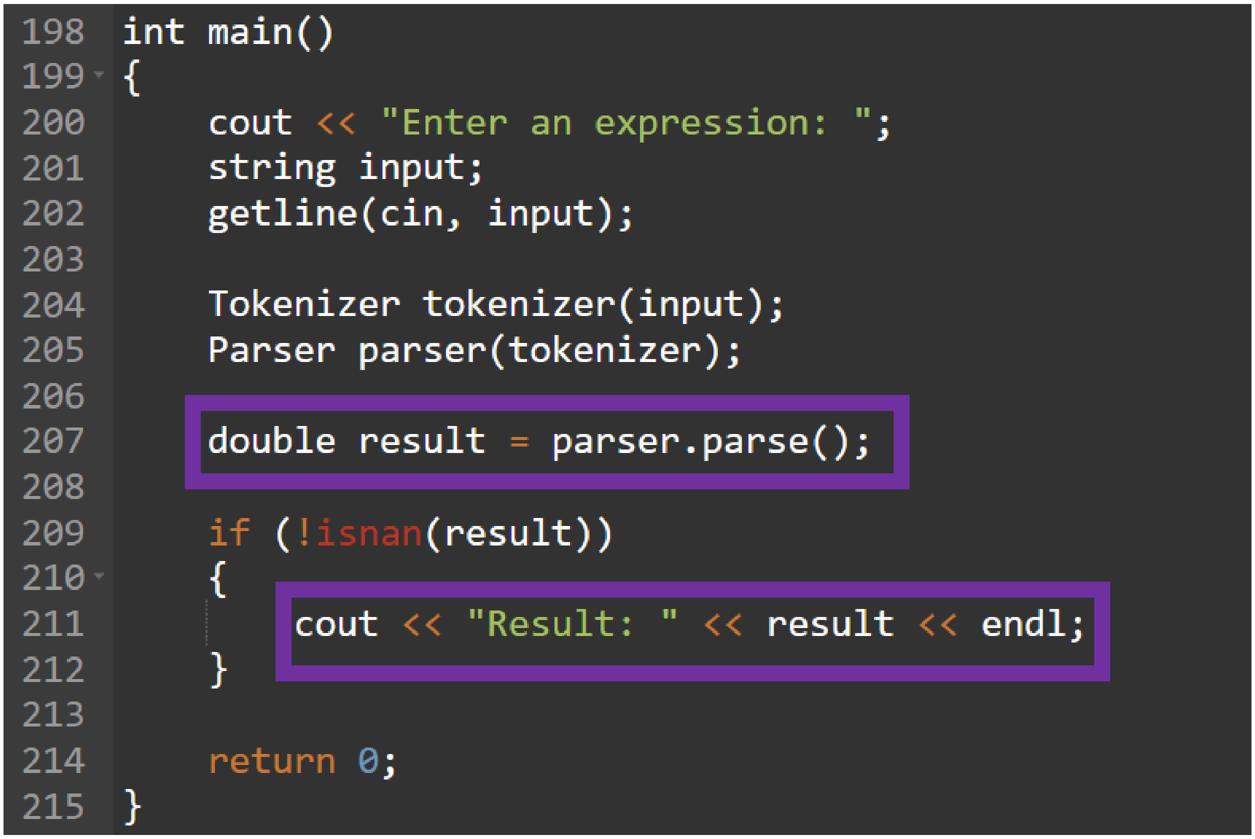
```
double expression()
{
    double left = term();
    while (currentToken_.type == TokenType::PLUS || currentToken_.type == TokenType::MINUS)
    {
        TokenType op = currentToken_.type;
        advanceToken();
        double right = term();
        if (op == TokenType::PLUS)
        {
            left += right;
        }
        else
        {
            left -= right;
        }
    }
    return left;
}
```

Parse() recieves our number and returns its



```
    double parse()
    {
        // Current Token is now the next token
        advanceToken();

        double result = expression();  // <- left off here 1
        if (currentToken_.type != TokenType::END)
        {
            cerr << "Unexpected token." << endl;
            return 0;
        }
        return result;
    }
```

This finally sends us back to main and outputs the result



```
int main()
{
    cout << "Enter an expression: ";
    string input;
    getline(cin, input);

    Tokenizer tokenizer(input);
    Parser parser(tokenizer);

    double result = parser.parse();

    if (!isnan(result))
    {
        cout << "Result: " << result << endl;
    }

    return 0;
}
```