C++ RDP Modifications and Tests

Zachary Smith

09 October 2019

CMSC 330 – Advanced Programming Languages

The first thing I noticed is that my compiler and the one in the material have exclusionary standards. For example, when trying to compile the source files included in the Module 5 readings, GCC complained when it encountered redundant class names in header files like so:

**variable.h:**

class Variable: public Operand {

public:

Variable(string name) {

this->name = name;

}

double Variable::evaluate(); // 🡨🡨🡨

private:

string name;

};

**Truncated output**:

$ g++ module3.cpp subexpression.cpp parse.cpp operand.cpp symboltable.cpp variable.cpp -o mid

variable.h:8:12: error: extra qualification ‘Variable::’ on member ‘evaluate’ [-fpermissive]

double Variable::evaluate();

[While this is permitted in Visual Basic (I did not confirm), GCC does not like it.](https://stackoverflow.com/questions/5642367/extra-qualification-error-in-c) Also, I was warned that the strstream header is deprecated, and I opted for string instead.

After adopting the recommendations for class Divide, I have added all the functionality except for the ternary condition. It became obvious that any reading from stdin should be done in parse(), not evaluate(), which necessitates inserting a third argument and instance variable to Subexpression, also as recommended. Output looks familiar to the week 7 discussion topic.

**Before moving input of the third argument from evaluate() to parse():**

$ ./mod3

Enter expression: (10:9?(a>3)),a=4;

Invoking Operand::parse() ...

Invoking Operand::parse() ...

Invoking Operand::parse() ...

left: 10

right: 9

question:

paren:

boolexpr: 0

Value = 9

**And after:**

$ ./mod3

Enter expression: (10:9?(a>3)),a=0;

Invoking Operand::parse() ...

Invoking Operand::parse() ...

Invoking Operand::parse() ...

Invoking Operand::parse() ...

Invoking Operand::parse() ...

left: 10

right: 9

boolexpr: 10

Value = 10

It turns out in a copy-and-paste error, I had been initializing the third argument **boolExpr** as **left** since making the changes, which have been subsequently corrected and tested.

$ ./mod3

Enter expression: ( ( ( z < ( 90 + aa ) ) & ( bb ! ) ) \* ( ( 7 / cc ) | ( 1 : 0 ? ( z > aa ) ) , z = 8 , aa = 2 , bb = 8 , cc = 2 ;

Value = 0

$ ./mod3

Enter expression: ( ( dd < ( 0 + ee ) ) : ( ff ! ) ? ( ( 7 - gg ) | ( 1 : 0 ? ( gg > dd ) ) , dd = 1 , ee = 3 , ff = 10 , gg = 7 ;

Value = 1

Good to know, but is this a multiple inheritance scenario? **Operand::parse()** is called twice in the vicinity of an operator, and I added an “if” statement in **SubExpression::parse()** that calls **Operand::parse()** a third time if the operator is a colon. If an open paren is encountered, the contents are treated like a new subexpression, repeating the process. To illustrate locations in the stream where **Operand::parse()** should be called:

**( 10 : 9 ? (a > 3) ) , a = 0;**

**^ ^ ^^ ^**

**1 2 34 5**

Furthermore, there are no inheritance clauses in the code I was provided that specify more than one base class. The abundance of **parse()** invocations can be explained by the recursive nature of the application, which looks obvious in retrospect.

I learned while testing that an assignment section is required (eg, … , a = 0;), but the variables need not exist in the expression body. We could have for example **(10>11),a=0;** where variable **a** has no bearing on the result but still exists in the symbol table.

Concerning the task of parsing multi-line file input – if this project were not platform-agnostic I would pipe a test file in to the applications stdin and call it a day. [xargs](http://man7.org/linux/man-pages/man1/xargs.1.html) would be thrown in to handle multiple lines.

$ cat input | tr '\n' '\0' | xargs -0 -n1 ./mod3

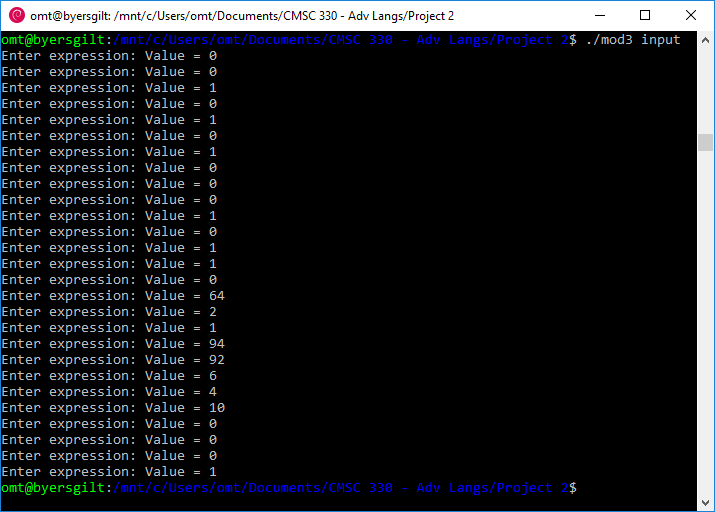
Enter expression: Invoking Operand::parse() ...

Invoking Operand::parse() ...

xargs: ./mod3: terminated by signal 11

Well that didn’t go as I had [hoped](https://stackoverflow.com/questions/199266/make-xargs-execute-the-command-once-for-each-line-of-input). Signal 11 is a segmentation fault. Unrelated – through this research I am excited to learn than that system call **pipe()** is used for inter-process communication since child processes created via **fork()** will inherit (maybe not the right word) open pipes of their parents. Linux has [**pipe()**](http://man7.org/linux/man-pages/man2/pipe.2.html)and Windows has [**CreatePipe()**,](https://docs.microsoft.com/en-us/windows/win32/ipc/anonymous-pipe-operations) and going further down this rabbit hole, while illuminating, is not helping to finish the project. We need something that is built in to C++. Hopefully the end result will also compile in Visual Studio. If not I recommend using [WSL2](https://docs.microsoft.com/en-us/windows/wsl/wsl2-index).

With [**std::ios::rdbuf()**](http://www.cplusplus.com/reference/ios/ios/rdbuf/), we can redirect a stream buffer to **cin**. Now we just need to split the file into lines and iteratively, call **cin.rdbuf()** on each line’s internal buffer before parsing. The original body of **main()** was moved in to **module3::parseLine()** for reuse.



My first test showed a segmentation fault after the last line of input, which is because I didn’t put the single-line **parseLine()** call inside of an else, in case no file name argument is supplied.

File ‘**input**’ was taken directly from Dr Waithe’s post regarding the Final Project, and the above output matches expected values:

|  |  |
| --- | --- |
| **Expression** | **Expected Value** **/ Success?** |
| ( a > 8 ) , a = 8 ; | 0 / yes |
| ( b > 8 ) , b = 7 ; | 0 / yes |
| ( c > 8 ) , c = 9 ; | 1 / yes |
| ( d < 5 ) , d = 5 ; | 0 / yes |
| ( e < 5 ) , e = 4 ; | 1 / yes |
| ( f < 5 ) , f = 6 ; | 0 / yes |
| ( g ! ) , g = 0 ; | 1 / yes |
| ( h ! ) , h = 1 ; | 0 / yes |
| ( i ! ) , i = 9 ; | 0 / yes |
| ( j & 7 ) , j = 0 ; | 0 / yes |
| ( k & 7 ) , k = 2 ; | 1 / yes |
| ( 0 | l ) , l = 0 ; | 0 / yes |
| ( 0 | m ) , m = 3 ; | 1 / yes |
| ( 1 | n ) , n = 3 ; | 1 / yes |
| ( o \* 8 ) , o = 0 ; | 0 / yes |
| ( p \* 8 ) , p = 8 ; | 64 / yes |
| ( q / 2 ) , q = 4 ; | 2 / yes |
| ( r / 2 ) , r = 2 ; | 1 / yes |
| ( 90 + s ) , s = 4 ; | 94 / yes |
| ( 90 + t ) , t = 2 ; | 92 / yes |
| ( 9 - u ), u = 3 ; | 6 / yes |
| ( 9 - v ), v = 5 ; | 4 / yes |
| ( 10 : 0 ? ( w & 3 ) ) , w = 3 ; | 10 / yes |
| ( 10 : 0 ? ( x & 3 ) ) , x = 0 ; | 0 / yes |
| ( 10 : 0 ? ( y & 0 ) ) , y = 0 ; | 0 / yes |
| ( ( ( z < ( 90 + aa ) ) & ( bb ! ) ) \* ( ( 7 / cc ) | ( 1 : 0 ? ( z > aa ) ) , z = 8 , aa = 2 , bb = 8 , cc = 2 ; | 0 / yes |
| ( ( dd < ( 0 + ee ) ) : ( ff ! ) ? ( ( 7 - gg ) | ( 1 : 0 ? ( gg > dd ) ) , dd = 1 , ee = 3 , ff = 10 , gg = 7 ; | 1 / yes |

Converting the operand values from double to int types was a matter of **ctrl+f** and **replace-all**. I was on the lookout for anything that should remain a double type but none were apparent. A follow-up test confirmed the same results as above. This is my compilation command:

**$ g++ module3.cpp subexpression.cpp parse.cpp operand.cpp symboltable.cpp variable.cpp divide.cpp minus.cpp times.cpp greaterthan.cpp lessthan.cpp equals.cpp bitand.cpp bitor.cpp negate.cpp condition.cpp -o mod3**