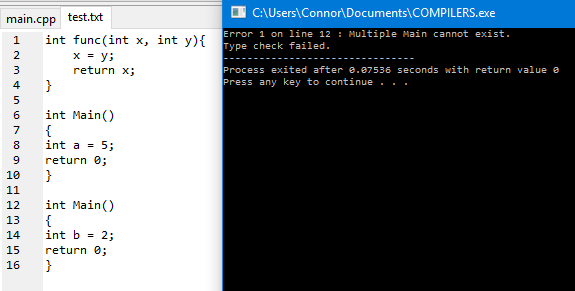
**Type Checking Project**

**By Connor Beckett-Lemus**

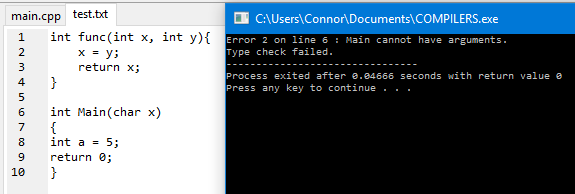
**Code Listing starts on Page 11**

**Error Code Demonstrations:**

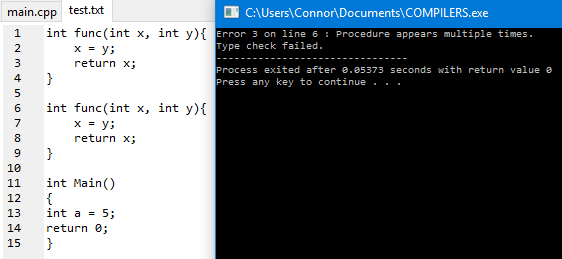
**Code 1:**



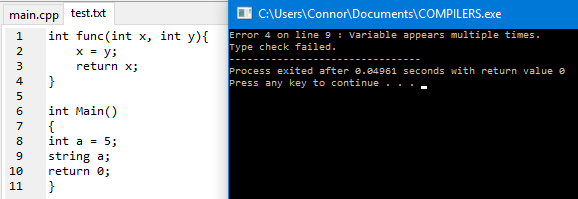
**Code 2:**



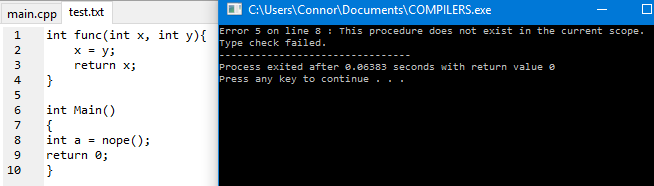
**Code 3:**



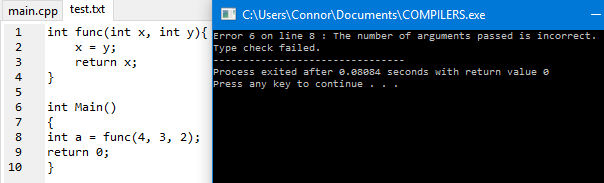
**Code 4:**



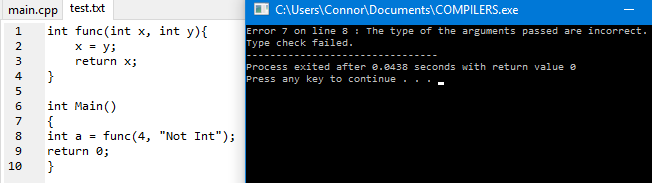
**Code 5:**



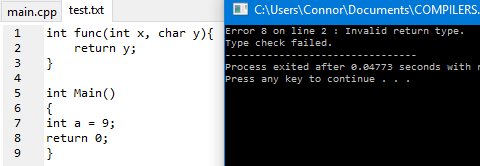
**Code 6:**

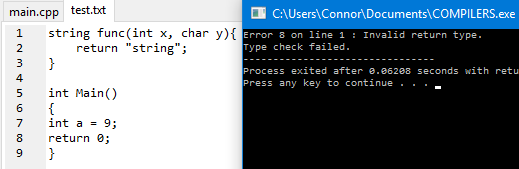


**Code 7:**

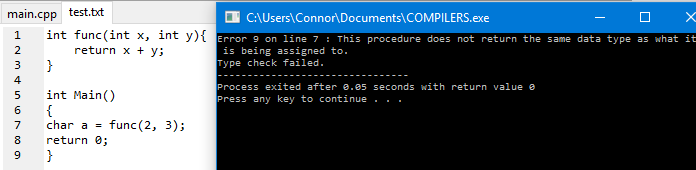


**Code 8:**

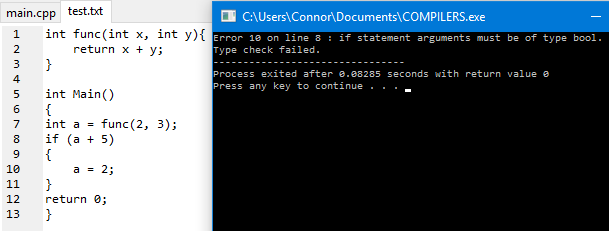




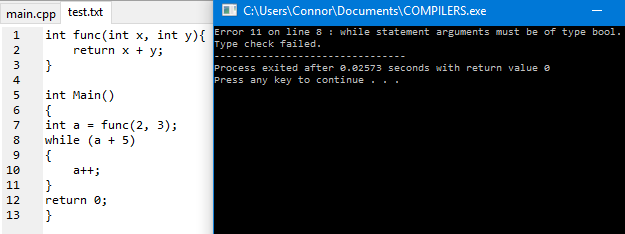
**Code 9:**



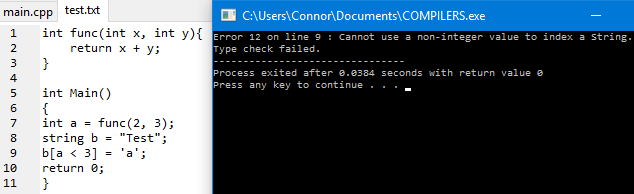
**Code 10:**



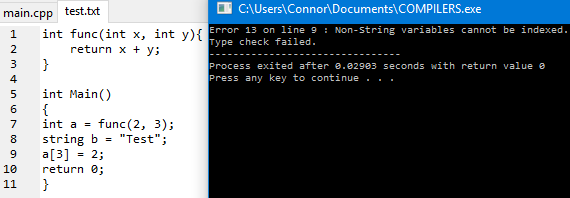
**Code 11:**



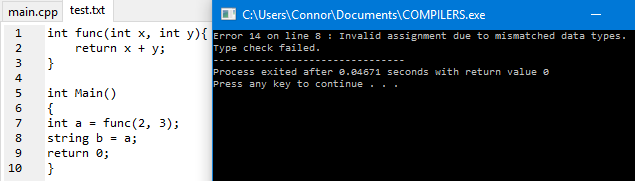
**Code 12:**

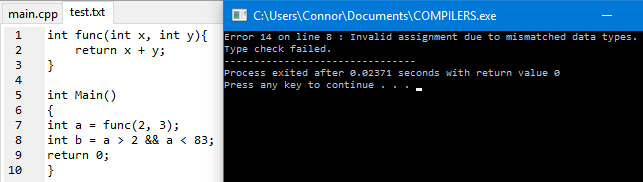


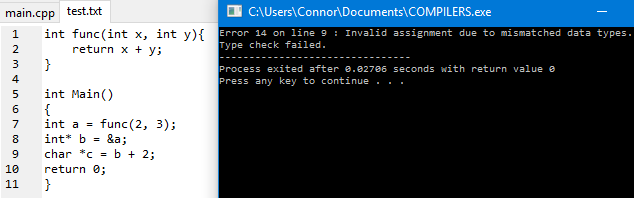
**Code 13:**

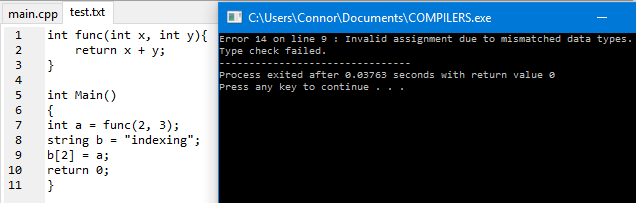


**Code 14:**

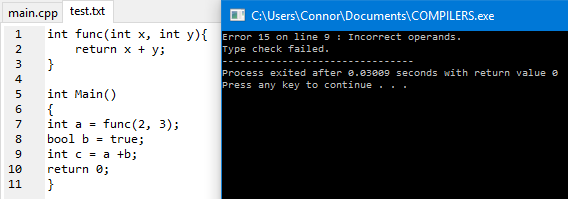


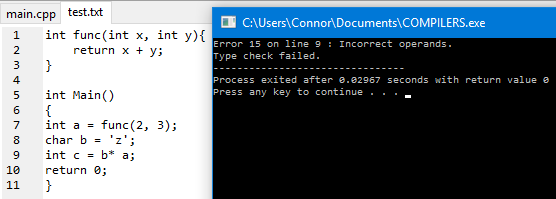


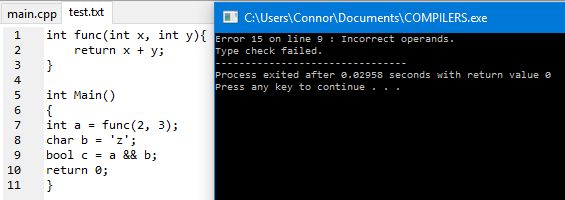


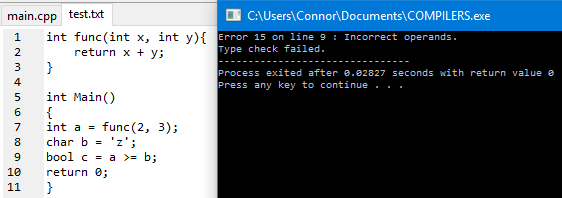


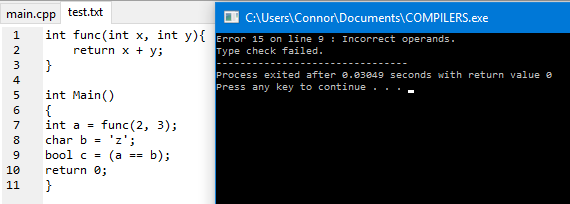
**Code 15:**

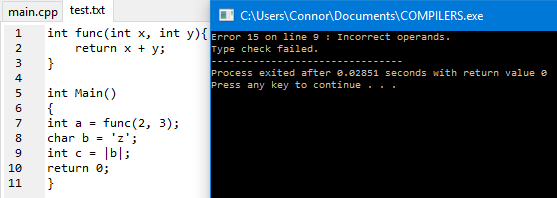


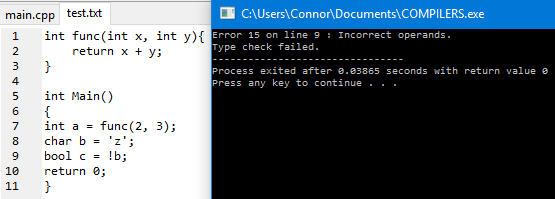




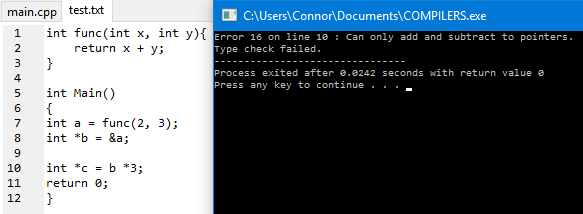




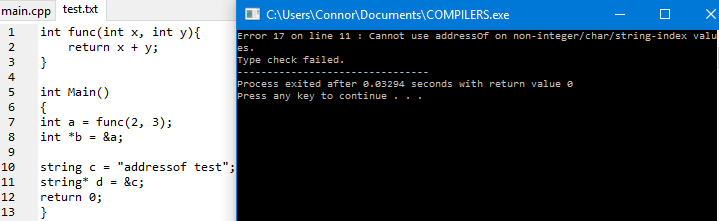




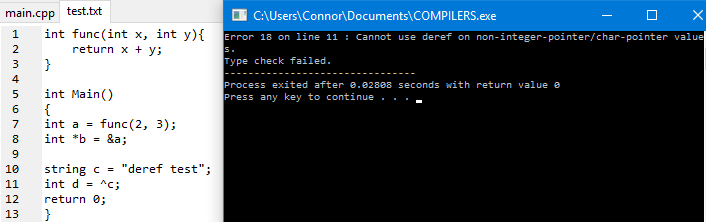
**Code 16:**



**Code 17:**

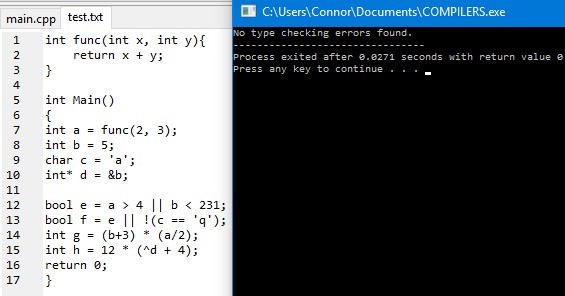


**Code 18:**

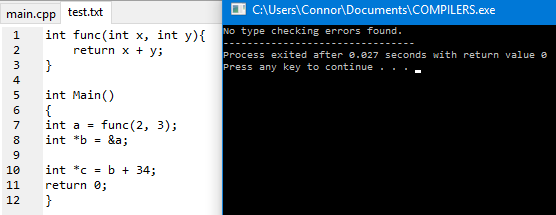


**Demonstration of valid input file:**

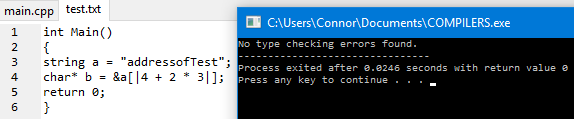
**Various expressions:**

****

**Adding to a pointer:**

****

**Using addressOf on an indexed string (and expression inside absolute value):**

****

**Code listing:**

#include <iostream>

#include <fstream>

#include <iomanip>

#include <unordered\_map> // need to add -std=c++11 under Tools->Compiler Options

#include <unordered\_set>

#include <vector>

using namespace std;

// A class that holds all necessary information about variables and functions from the input file.

class symbolInfo

{

int scope; // the scope this symbol was initially declared on

string type; // the data type of this symbol

vector<string> arguments; // the list of arguments the symbol accepts if it's a function

public:

symbolInfo(int s, string t) : scope(s), type(t) {}

symbolInfo(int s, string t, vector<string> a) : scope(s), type(t), arguments(a) {}

string getType() {return type;}

int getScope() {return scope;}

vector<string> getArguments() {return arguments;}

};

// The current scope of the file

int fileScope = 0;

// The current line number of the file

int lineNo = 1;

// The current function the checker is in

string currentFunc;

// The table (as a hash map) that holds all the data about the variables and functions, using their names as keys.

unordered\_map<string, symbolInfo\*> symbolTable;

// A set of keywords.

unordered\_set<string> keywords =

{"int", "char", "double", "short", "long", "void", "class", "switch", "case", "bool", "float", "string", "return", "break", "if", "else", "while", "for", "true", "false"};

// A set of operators mostly, with whitespace characters added to assist in scanning.

unordered\_set<char> operators = {'+', '-', '\*', '/', '=', '<', '>', '!', '.', '(', ')', '{', '}', ';', '^', '%', ':', ' ', ',', '\n', '\t', '?', '[', ']', '&', '|'};

// A set containing all the operators that are 2 characters.

unordered\_set<string> twoCharOps = {"&&", "||", "==", "<=", "!=", "+=", "-=", "\*=", "/=", "->", "++", "--", "<<", ">>", "::"};

// A set containing all of the data types.

unordered\_set<string> types = {"int", "char", "double", "float", "short", "long", "void", "bool", "string"};

// Breaks an input file into a vector of tokens.

// Preconditions: An empty string vector.

// Postconditions: The passed vector is filled with the input's tokens.

void breakTokens(vector<string>\*);

// Checks the tokens from the passed vector to determine if their are any type errors in the program.

// Preconditions: The vector is filled with valid Csimple tokens.

// Postconditions: None.

void typeCheck(vector<string>);

// Takes in an expression (composed of tokens) and determines the data type.

// Returns a string representation of the data type, or "e" if the expression had a type error.

// Preconditions: The vector is filled with valid Csimple tokens.

// Postconditions: None.

string parseExpression(vector<string>);

// Determines the type of the passed token.

// Returns a string representation of the data type.

// Preconditions: None.

// Postconditions: None.

string tokenType(string\*);

// Finds the first token in the passed vector equal to the passed string.

// Returns the index the token was found at, or -1 if the token was not found.

// Preconditions: None

// Postconditions: None

int findFirst(vector<string>, string);

// Determines if the passed function (composed of tokens) is a valid function call.

// Returns true if it is a valid function call, false otherwise.

// Preconditions: The vector is filled with valid Csimple tokens.

// Postconditions: None.

bool functionCheck(vector<string>);

// Displays error information when an error is encountered.

// Preconditions: None.

// Postconditions: An error message appears in the console.

void showError(int);

int main()

{

vector<string> tokens;

breakTokens(&tokens);

typeCheck(tokens);

return 0;

}

void breakTokens(vector<string>\* tokenList)

{

ifstream input("test.txt");

string word = "";

bool isString = false;

bool isChar = false;

bool isNumber = false;

char current;

char previous;

while (!input.eof())

{

current = input.get();

if (operators.find(current) == operators.end() || isString || isChar || (isNumber && current == '.')) // current character is not an operator (building word)

{

word += current;

if (current == '\"')

{

isString = !isString;

}

else if (current == '\'')

{

isChar = !isChar;

}

else if (current > 47 && current < 58) // current is a number (ascii range 48-57)

{

isNumber = true;

}

}

else // if the current char is an operator and the scanner is not currently building a string, char, or number

{

string op = string(1, current);

if (word != "") // add the word that was being built up before the op was reached

{

tokenList->push\_back(word);

}

if (twoCharOps.find(previous + op) != twoCharOps.end())

{

tokenList->pop\_back();

tokenList->push\_back(previous + op);

}

else

{

if (current != ' ' && current != '\t') // whitespace is not a token

{

tokenList->push\_back(op);

}

}

word = "";

isString = false;

isChar = false;

if (current != '.')

{

isNumber = false;

}

}

previous = current;

}

input.close();

}

void typeCheck(vector<string> tokens)

{

string lastValue;

for (int i = 0; i < tokens.size(); i++)

{

string current = tokens[i];

if (current == "{")

{

fileScope++;

}

else if (current == "}")

{

fileScope--;

}

else if (current == "\n")

{

lineNo++;

}

else if (types.find(current) != types.end()) // current token is a data type - line is a declaration

{

string pointerAdd = "";

if (tokens[i + 1] == "\*") // pointer

{

i++;

pointerAdd = "\*";

}

string name = tokens[i + 1];

unordered\_map<string, symbolInfo\*>::iterator it = symbolTable.find(name);

string next = tokens[i + 2];

vector<string> arguments;

if (next == "(") // function

{

if (current == "string")

{

showError(8);

return;

}

if (name == "Main")

{

if (it != symbolTable.end() || fileScope != 0) // main already exists in symbol table or not currently in global scope

{

showError(1);

return;

}

else if (tokens[i + 2] == "(" && tokens[i + 3] != ")")

{

showError(2);

return;

}

}

currentFunc = name;

if (it != symbolTable.end() && (it->second)->getScope() == fileScope) // duplicate function

{

showError(3);

return;

}

i += 2; // advancing to the arguments

int intoArgs = 0;

while (next != ")")

{

intoArgs++;

next = tokens[i + intoArgs];

if (types.find(next) != types.end())

{

arguments.push\_back(next);

}

}

symbolInfo\* info = new symbolInfo(fileScope, current, arguments);

pair<string, symbolInfo\*> data(name, info);

symbolTable.insert(data);

}

else // identifier

{

i++;

if (it != symbolTable.end() && (it->second)->getScope() == fileScope) // duplicate id

{

showError(4);

return;

}

symbolInfo\* info = new symbolInfo(fileScope, pointerAdd + current);

pair<string, symbolInfo\*> data(name, info);

symbolTable.insert(data);

}

}

else if (tokens[i + 1] == "(" && operators.find(current[0]) == operators.end() && keywords.find(current) == keywords.end()) // function calls

{

if (symbolTable.find(current) != symbolTable.end())

{

vector<string> function;

function.push\_back(current); // function name

while (current != ")")

{

i++;

current = tokens[i];

function.push\_back(current);

}

bool valid = functionCheck(function);

if (!valid)

{

return;

}

}

else // function not in symbol table

{

showError(5);

return;

}

}

else if (current == "[") // checking if indexing is being applied to a string and if the argument is an integer

{

if (tokenType(&tokens[i - 1]) != "string")

{

showError(13);

return;

}

vector<string> expression;

i++;

current = tokens[i];

while (current != "]")

{

expression.push\_back(current);

i++;

current = tokens[i];

}

string expressionType = parseExpression(expression);

if (expressionType == "e")

{

return;

}

else if (expressionType != "int")

{

showError(12);

return;

}

}

else if (current == "=") // assignment

{

string lhsType = tokenType(&tokens[i - 1]);

if (tokens[i + 2] == "(") // assigning a function to a value

{

if (symbolTable.find(tokens[i + 1]) == symbolTable.end()) // function not found

{

showError(5);

return;

}

if (tokenType(&tokens[i + 1]) != lhsType) // mismatched types

{

showError(9);

return;

}

}

else // assigning an expression

{

vector<string> expression;

i++;

current = tokens[i];

while (current != ";")

{

expression.push\_back(current);

i++;

current = tokens[i];

}

string rhsType = parseExpression(expression);

if (rhsType == "e")

{

return;

}

else if (rhsType != lhsType

&& !(rhsType == "\*null" && (lhsType == "\*int" || lhsType == "\*char"))) // allows null pointer to be assigned to int/char pointer

{

showError(14);

return;

}

}

}

else if (current == "return")

{

vector<string> expression;

i++;

string next = tokens[i];

while (next != ";")

{

expression.push\_back(next);

i++;

next = tokens[i];

}

string returnType = parseExpression(expression);

unordered\_map<string, symbolInfo\*>::iterator it = symbolTable.find(currentFunc);

if (returnType == "e")

{

return;

}

else if ((it->second)->getType() != returnType)

{

showError(8);

return;

}

}

else if (current == "if" || current == "while")

{

vector<string> expression;

i += 2;

string next = tokens[i];

while (next != ")")

{

expression.push\_back(next);

i++;

next = tokens[i];

}

string loopCondition = parseExpression(expression);

if (loopCondition == "e")

{

return;

}

else if (loopCondition != "bool")

{

if (current == "if")

{

showError(10);

}

else

{

showError(11);

}

return;

}

}

}

cout << "No type checking errors found.";

}

string parseExpression(vector<string> expression)

{

// This function goes through all the expression operators (and parentheses) in

// their appropriate evaluation order. It resolves each operator by removing all

// tokens used by the operator, then replaces the operator with a literal

// value of the operator's output type.

// ex. The sequence 4, <, 2 will have 4 and 2 removed, then < replaced with true.

// For literal pointers, \* is attached to the appropriate literal value to denote it.

int foundLoc = findFirst(expression, "(");

while (foundLoc != -1)

{

vector<string> subExpr;

string current = expression[foundLoc + 1];

while (current != ")")

{

subExpr.push\_back(current);

expression.erase(expression.begin() + foundLoc + 1);

current = expression[foundLoc + 1];

}

string subExprType = parseExpression(subExpr);

if (subExprType == "e")

{

return "e";

}

string pointerAdd;

string dummyValue;

if (subExprType[0] == '\*')

{

pointerAdd = "\*";

subExprType = subExprType.substr(1, subExprType.size());

}

if (subExprType == "char")

{

dummyValue = "\'";

}

else if (subExprType == "int")

{

dummyValue = "0";

}

else if (subExprType == "string")

{

dummyValue = "\"";

}

else if (subExprType == "bool")

{

dummyValue = "true";

}

expression[foundLoc] = pointerAdd + dummyValue;

expression.erase(expression.begin() + foundLoc + 1); // erasing ")"

foundLoc = findFirst(expression, "(");

}

foundLoc = findFirst(expression, "[");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) != "string")

{

showError(13);

return "e";

}

vector<string> subExpr;

string current = expression[foundLoc + 1];

while (current != "]")

{

subExpr.push\_back(current);

expression.erase(expression.begin() + foundLoc + 1);

current = expression[foundLoc + 1];

}

string expressionType = parseExpression(subExpr);

if (expressionType == "e")

{

return "e";

}

else if (expressionType != "int")

{

showError(12);

return "e";

}

expression[foundLoc] = "\'";

expression.erase(expression.begin() + foundLoc + 1); // erasing "]"

expression.erase(expression.begin() + foundLoc - 1); // erasing string id

foundLoc = findFirst(expression, "[");

}

foundLoc = findFirst(expression, "|");

while (foundLoc != -1)

{

vector<string> subExpr;

string current = expression[foundLoc + 1];

while (current != "|")

{

subExpr.push\_back(current);

expression.erase(expression.begin() + foundLoc + 1);

current = expression[foundLoc + 1];

}

string subExprType = parseExpression(subExpr);

if (subExprType == "e")

{

return "e";

}

else if (subExprType != "int")

{

showError(15);

return "e";

}

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1); // erasing closing "|"

foundLoc = findFirst(expression, "|");

}

foundLoc = findFirst(expression, "&");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc + 1]) == "int")

{

expression[foundLoc] = "\*0";

expression.erase(expression.begin() + foundLoc + 1);

}

else if (tokenType(&expression[foundLoc + 1]) == "char") // indexed strings just evaluate to char

{

expression[foundLoc] = "\*\'";

expression.erase(expression.begin() + foundLoc + 1);

}

else

{

showError(17);

return "e";

}

foundLoc = findFirst(expression, "&");

}

foundLoc = findFirst(expression, "^");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc + 1]) == "\*int")

{

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1);

}

else if (tokenType(&expression[foundLoc + 1]) == "\*char")

{

expression[foundLoc] = "\'";

expression.erase(expression.begin() + foundLoc + 1);

}

else

{

showError(18);

return "e";

}

foundLoc = findFirst(expression, "^");

}

foundLoc = findFirst(expression, "!");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc + 1]) == "bool")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "!");

}

foundLoc = findFirst(expression, "\*");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if (leftToken == "int" && rightToken == "int")

{

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else if (leftToken[0] == '\*' || rightToken[0] == '\*')

{

showError(16);

return "e";

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "\*");

}

foundLoc = findFirst(expression, "/");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if (leftToken == "int" && rightToken == "int")

{

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else if (leftToken[0] == '\*' || rightToken[0] == '\*')

{

showError(16);

return "e";

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "/");

}

foundLoc = findFirst(expression, "+");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if (leftToken == "int" && rightToken == "int")

{

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else if ((leftToken[0] == '\*' && rightToken == "int")

|| (leftToken == "int" && rightToken[0] == '\*'))

{

string dummyValue;

string pointerType;

if (leftToken[0] == '\*')

{

pointerType = leftToken.substr(1, leftToken.size());

}

else // right token is the pointer

{

pointerType = rightToken.substr(1, rightToken.size());

}

if (pointerType == "char")

{

dummyValue = "\'";

}

else if (pointerType == "int")

{

dummyValue = "0";

}

else if (pointerType == "string")

{

dummyValue = "\"";

}

else if (pointerType == "bool")

{

dummyValue = "true";

}

expression[foundLoc] = "\*" + dummyValue;

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "+");

}

foundLoc = findFirst(expression, "-");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if (leftToken == "int" && rightToken == "int")

{

expression[foundLoc] = "0";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else if (leftToken[0] == '\*' && rightToken == "int")

{

string dummyValue;

string pointerType = leftToken.substr(1, leftToken.size());

if (pointerType == "char")

{

dummyValue = "\'";

}

else if (pointerType == "int")

{

dummyValue = "0";

}

else if (pointerType == "string")

{

dummyValue = "\"";

}

else if (pointerType == "bool")

{

dummyValue = "true";

}

expression[foundLoc] = "\*" + dummyValue;

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "-");

}

foundLoc = findFirst(expression, "<");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "int" && tokenType(&expression[foundLoc + 1]) == "int")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "<");

}

foundLoc = findFirst(expression, ">");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "int" && tokenType(&expression[foundLoc + 1]) == "int")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, ">");

}

foundLoc = findFirst(expression, "<=");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "int" && tokenType(&expression[foundLoc + 1]) == "int")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "<=");

}

foundLoc = findFirst(expression, ">=");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "int" && tokenType(&expression[foundLoc + 1]) == "int")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, ">=");

}

foundLoc = findFirst(expression, "==");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if ((leftToken == "int" && rightToken == "int")

|| (leftToken == "char" && rightToken == "char")

|| (leftToken == "bool" && rightToken == "bool")

|| ((leftToken == "\*int" || leftToken == "\*null") && (rightToken == "\*int" || rightToken == "\*null"))

|| ((leftToken == "\*char" || leftToken == "\*null") && (rightToken == "\*char" || rightToken == "\*null")))

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "==");

}

foundLoc = findFirst(expression, "!=");

while (foundLoc != -1)

{

string leftToken = tokenType(&expression[foundLoc - 1]);

string rightToken = tokenType(&expression[foundLoc + 1]);

if ((leftToken == "int" && rightToken == "int")

|| (leftToken == "char" && rightToken == "char")

|| (leftToken == "bool" && rightToken == "bool")

|| ((leftToken == "\*int" || leftToken == "\*null") && (rightToken == "\*int" || rightToken == "\*null"))

|| ((leftToken == "\*char" || leftToken == "\*null") && (rightToken == "\*char" || rightToken == "\*null")))

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "!=");

}

foundLoc = findFirst(expression, "&&");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "bool" && tokenType(&expression[foundLoc + 1]) == "bool")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "&&");

}

foundLoc = findFirst(expression, "||");

while (foundLoc != -1)

{

if (tokenType(&expression[foundLoc - 1]) == "bool" && tokenType(&expression[foundLoc + 1]) == "bool")

{

expression[foundLoc] = "true";

expression.erase(expression.begin() + foundLoc + 1);

expression.erase(expression.begin() + foundLoc - 1);

}

else

{

showError(15);

return "e";

}

foundLoc = findFirst(expression, "||");

}

return tokenType(&expression[0]);

}

string tokenType(string\* token)

{

unordered\_map<string, symbolInfo\*>::iterator it = symbolTable.find(\*token);

char first = (\*token)[0];

string pointerAdd = "";

if (first == '\*') // pointer "literal" signifier, can't exist in the input file but is used in the expression parser as a dummy value

{

pointerAdd = "\*";

\*token = token->substr(1, token->size());

first = (\*token)[0];

}

if (it != symbolTable.end()) // if in symbol table

{

return (it->second)->getType();

}

else if (token->find("\"") != string::npos) // if not in symbol table, must be a literal to be valid

{

return pointerAdd + "string";

}

else if (token->find("\'") != string::npos || \*token == "]")

{

return pointerAdd + "char";

}

else if (first > 47 && first < 58)

{

if (token->find(".") != string::npos)

{

return pointerAdd + "double";

}

else

{

return pointerAdd + "int";

}

}

else if (\*token == "true" || \*token == "false")

{

return pointerAdd + "bool";

}

else

{

if (pointerAdd == "\*")

{

return "\*null"; // null pointer

}

return "void";

}

}

int findFirst(vector<string> vect, string search)

{

for (int i = 0; i < vect.size(); i++)

{

if (vect[i] == search)

{

return i;

}

}

return -1; // search not found

}

bool functionCheck(vector<string> function)

{

unordered\_map<string, symbolInfo\*>::iterator it = symbolTable.find(function[0]);

if (it != symbolTable.end() && (it->second)->getScope() <= fileScope)

{

vector<string> arguments;

string next = function[2]; // start past the first ( of the function call

int i = 0;

while (next != ")")

{

if (next != ",")

{

arguments.push\_back(tokenType(&next));

}

i++;

next = function[2 + i];

}

vector<string> storedArguments = (it->second)->getArguments();

if (arguments.size() != storedArguments.size())

{

showError(6);

return false;

}

for (i = 0; i < arguments.size(); i++)

{

if (arguments[i] != storedArguments[i])

{

showError(7);

return false;

}

}

}

else // function does not exist in the current scope

{

showError(5);

return false;

}

return true;

}

void showError(int code)

{

cout << "Error " << code << " on line " << lineNo << " : ";

switch(code)

{

case 1:

cout << "Multiple Main cannot exist." << endl;

break;

case 2:

cout << "Main cannot have arguments." << endl;

break;

case 3:

cout << "Procedure appears multiple times." << endl;

break;

case 4:

cout << "Variable appears multiple times." << endl;

break;

case 5:

cout << "This procedure does not exist in the current scope." << endl;

break;

case 6:

cout << "The number of arguments passed is incorrect." << endl;

break;

case 7:

cout << "The type of the arguments passed are incorrect." << endl;

break;

case 8:

cout << "Invalid return type." << endl;

break;

case 9:

cout << "This procedure does not return the same data type as what it is being assigned to." << endl;

break;

case 10:

cout << "if statement arguments must be of type bool." << endl;

break;

case 11:

cout << "while statement arguments must be of type bool." << endl;

break;

case 12:

cout << "Cannot use a non-integer value to index a String." << endl;

break;

case 13:

cout << "Non-String variables cannot be indexed." << endl;

break;

case 14:

cout << "Invalid assignment due to mismatched data types." << endl;

break;

case 15:

cout << "Incorrect operands." << endl;

break;

case 16:

cout << "Can only add and subtract to pointers." << endl;

break;

case 17:

cout << "Cannot use addressOf on non-integer/char/string-index values." << endl;

break;

case 18:

cout << "Cannot use deref on non-integer-pointer/char-pointer values." << endl;

break;

default:

cout << "Undefined error." << endl;

break;

}

cout << "Type check failed.";

}