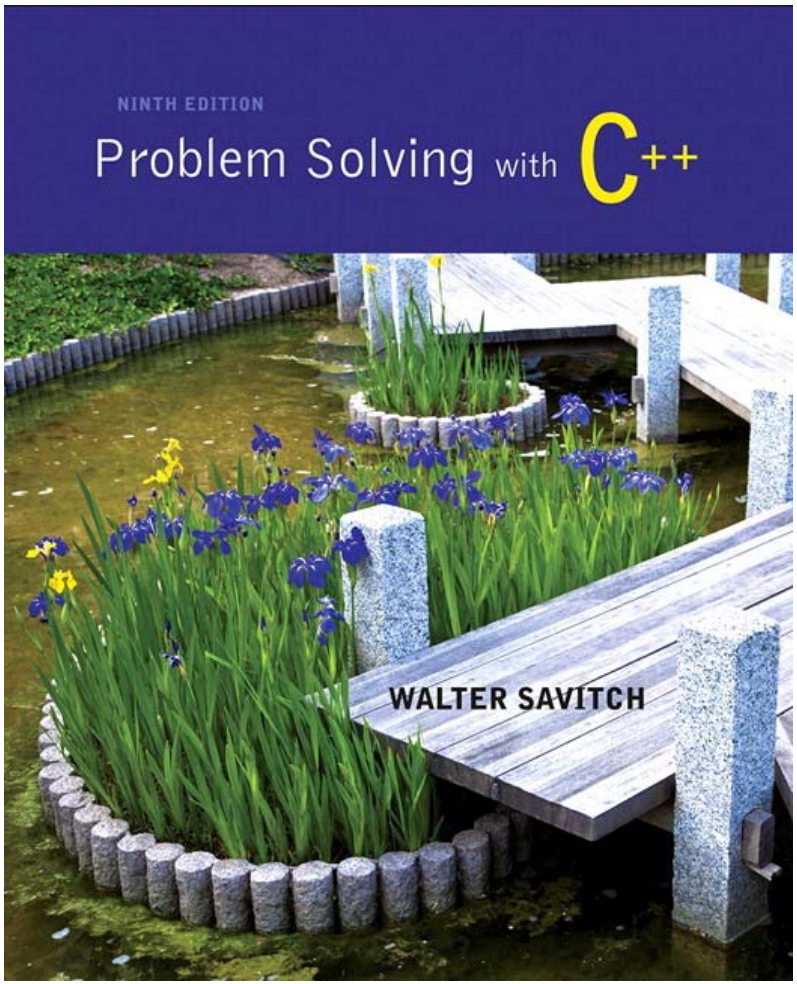
Review: Expressions, Variables, Loops, and more.



An Expression Evaluator Example [2]

Case Study : Parsing PostFix Expressions

What is an expression?

- A series of symbols that return some value when evaluated
- Syntax rules determine if an expression is valid or not
- Operators act on expressions to create new expressions
- C++ has lots of expressions and expression operators
  - Arithmetic : 1+3, 3-4, 5*6, 15/10, 15%10, 15/10.0
  - Variables : a, b, myVar, Fred, argv
  - Logical : a&&b, a||b (What do these equal if a=1 and b=0?)
  - Relational : a<b, a>b, a==b, a!=b, a<b

The Expression Parsing Problem:

- Write a program that evaluates arithmetic expressions
- For example: what does 43.2 / (100 + 5) evaluate to?
- We call such programs an Expression Evaluator
- Evaluating C++ expressions is a tricky problem!
- Reverse Polish Notation (RPN) or PostFix is much simpler!

PostFix Expressions

C++ arithmetic expressions like 5+2 are in Infix notation

- That means that binary operators go in-between the operands

PostFix Expressions place the operator after the operands

- So in PostFix, 5+2 is expressed as: 5 2 +

As few simple examples:

1 2 + is 3, because it means 1+2
4 7 – is -3, because it means 4-7
3 4 * is 12, because it means 3*4
8 4 / is 2, because it means 8/4

We will write a PostFix Expression Evaluator in C++

Evaluating PostFix Expressions

It gets more complex when there is more than one operator. How would we write the infix expression 1 + 2 * 3 in postfix?

Since * is always evaluated before + with infix, we get this postfix expression: 2 3 * 1 +

You evaluate starting at the left, and apply an operator to the two operands immediately proceeding it.

2 3 * 1 +

→ 6 1 +

→ 7

After evaluating an operator, you replace the numbers and operand with the new value.

More PostFix Example Expressions

Now let's write the infix expression **(1 + 2) * 3** in postfix
This time **+** is evaluated before ***** because of the brackets

The postfix expression is:

1 2 + 3 *

It is evaluated in these steps:

1 2 + 3 * → 3 3 * → 9

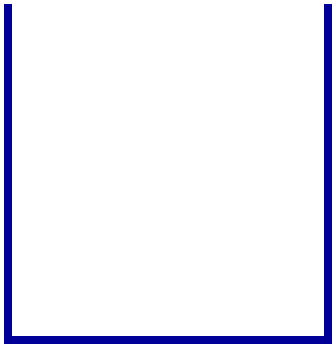
The area of a circle πr^2 is written: **pi * r * r** in infix notation

In postfix, it could be written as **pi r * r *** or **r r * pi ***

Stacks

To evaluate a postfix expression, we need to introduce the idea of a **stack**, which is a simple data structure that lets us add and remove items at the top of the stack only.

Pictorially, we'll draw a stack as a box with no top
Here's an empty stack:



Top of Stack

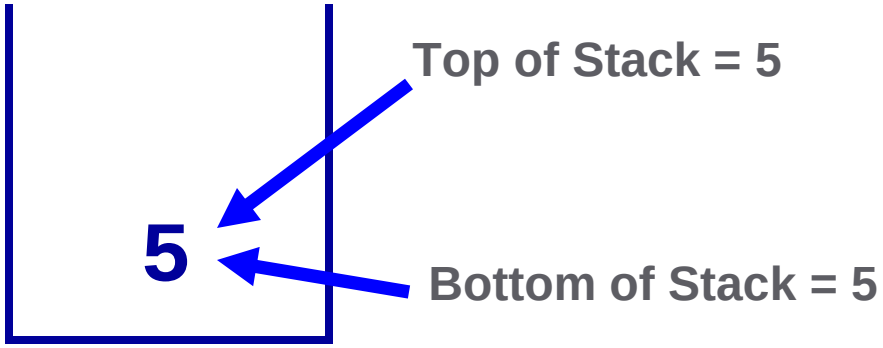
Bottom of Stack

Empty Stack

Pushing Items Onto a Stack

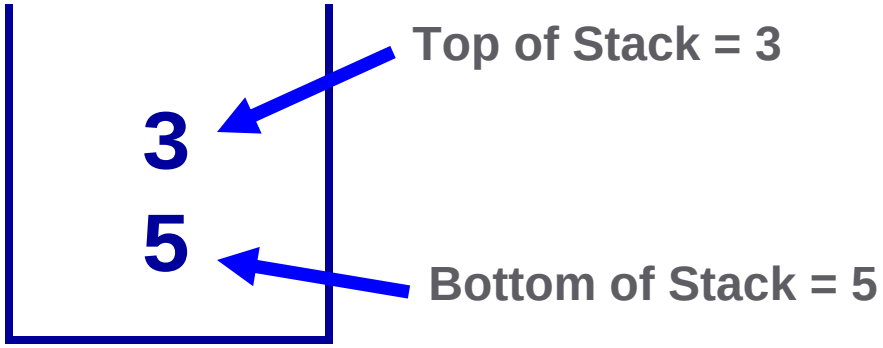
Adding an element to a stack is called *Pushing* an element
Function **`push()`** is used to push an element onto the stack

For example:
Here is what the stack looks like after we push 5 onto it:

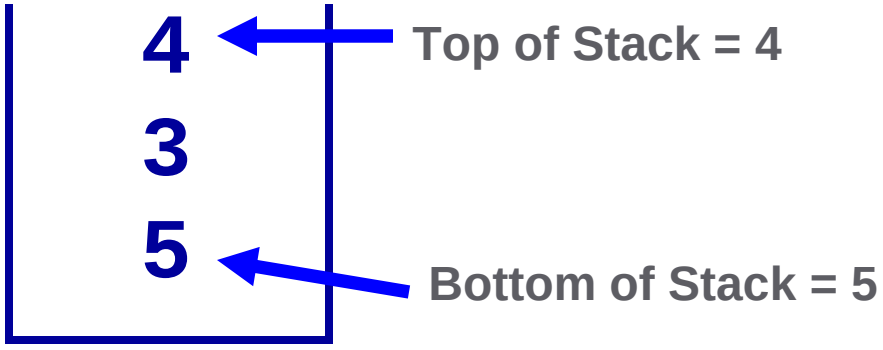


Pushing Always Adds Elements to the Top

After pushing 3 onto a stack already containing 5:



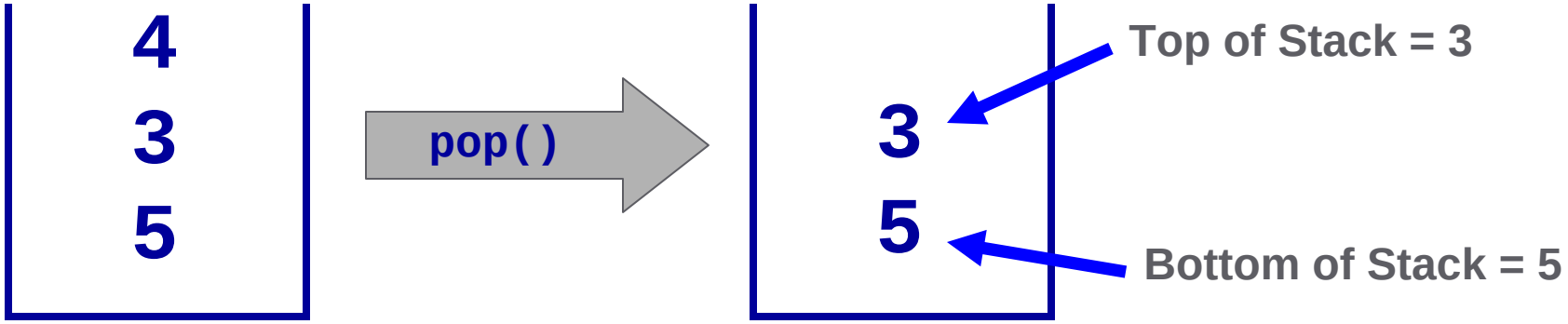
Whenever you push an element, it becomes the new Top. Let's push 4:



Popping Items from a Stack

Can only remove the top element of a stack
This is called **popping** the stack
Use the function name **pop()** to pop the stack

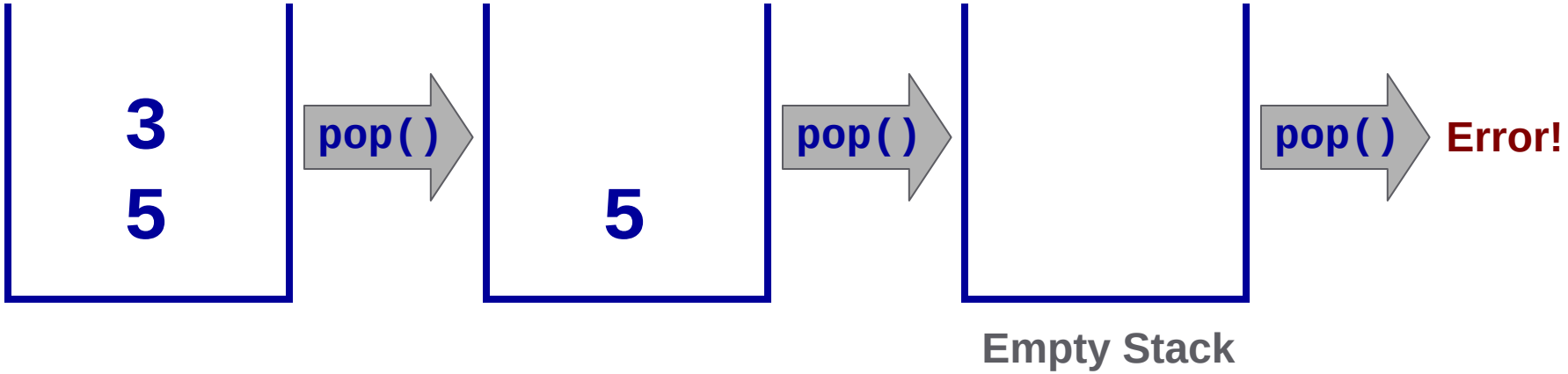
If we pop our stack then it looks like this:



Pop removed 4 from the stack

Popping from an Empty Stack

Popping the Stack again : 3 is removed
Popping the Stack 3rd time: 5 is removed, the stack is empty
Popping an empty stack is an error!



Using a `vector` as a Stack

Many different ways to implement a stack
We will implement a stack using a **`vector`**
`vector` was designed with stacks in mind!
`push_back` and **`pop_back`** methods work like a stack

Example:

```
vector<int> stack; // initially empty

stack.push_back(5); // push 5
stack.push_back(3); // push 3
stack.push_back(4); // push 4

cout <<"top of stack is " << stack.back(); // prints 4

stack.pop_back(); // 4 is popped
stack.pop_back(); // 3 is popped
stack.pop_back(); // 5 is popped
```

Stacks as a Vector

A Stack is a data structure with three operations:

- **push(x)** puts **x** on the top of the stack
- **pop()** removes the top element of the stack
- **peek()** returns a copy of the top element *without* removing it

Can treat **stack** as a **vector** and use any **vector** operation on it

With stacks, normally when pop is called, we want to view the top element

- **stack.pop_back()** does *not* return the popped element
- all it does is delete the top of the stack
- so before calling **pop_back** we usually ***peek*** at the top using **stack.back()**

The Postfix Expression Evaluation Algorithm

How do you evaluate the postfix expression **3 10 + 2 *** ?
Here is the complete algorithm in pseudo-code:

```
stack is initially empty
for each token in the expression do:
    if token is a number then
        push it onto the stack
    else if token is "+" then
        pop the top 2 elements of the stack; call them a and b
        push a + b onto the stack
    else if token is "*" then
        pop the top 2 elements of the stack; call them a and b
        push a * b onto the stack
    else if token is "-" then
        pop the top 2 elements of the stack; call them a and b
        push a - b onto the stack
    else if token is "/" then
        pop the top 2 elements of the stack; call them a and b
        push a / b onto the stack
end for
```


Slides based on Problem Solving with C++, 9th Edition, Walter Savitch
and Course Notes on Expression Evaluator by Dr. Toby Donaldson

The Postfix Expression Evaluation Algorithm

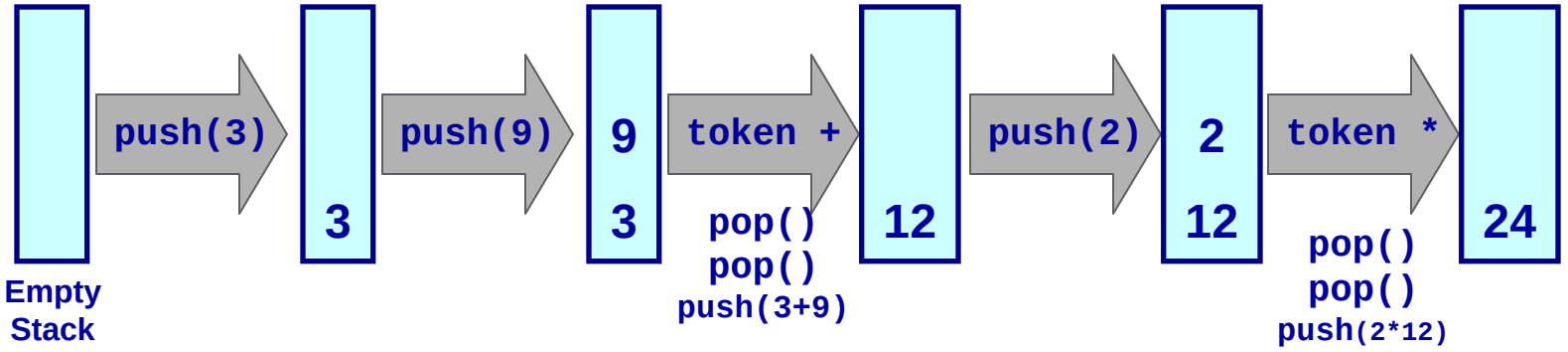
When this loop is done, the value of the expression is the number on the top of the stack
A **token** is either a number (e.g. 10) or an operator (e.g. ***)
For example, the expression **3 10 + 2 *** has 5 tokens: three operands and two operators

```
stack is initially empty
for each token in the expression do:
    if token is a number then
        push it onto the stack
    else if token is "+" then
        pop the top 2 elements of the stack; call them a and b
        push a + b onto the stack
    else if token is "*" then
        pop the top 2 elements of the stack; call them a and b
        push a * b onto the stack
    else if token is "-" then
        pop the top 2 elements of the stack; call them a and b
        push a - b onto the stack
    else if token is "/" then
        pop the top 2 elements of the stack; call them a and b
        push a / b onto the stack
end for
```

A Traced Example

Let's trace through the evaluation of expression: 3 9 + 2 *

- 3 is read: it is pushed onto the stack
- 9 is read: it is pushed onto the stack
- **+** is read, so 9 and 3 are popped, setting a = 3 and b = 9.
  Then a + b, which is 12, is pushed onto the stack
- 2 is read: it is pushed onto the stack
- * is read, so 2 and 12 are popped, setting a = 12 and b = 2
  Then a * b, which is 24, is pushed onto the stack
- There are no more tokens, so the value of the postfix expression is the number on the top of the stack: 24



PostFix Practice

What do each of the following postfix expressions evaluate to?

		Solutions:
1.	4 2 +	6
2.	4 2 -	2
3.	4 3 * 5 +	17
4.	4 3 + 5 *	35
5.	2 2 * 4 4 * + 4 /	5

Writing a C++ Postfix Evaluator

Now that we know how to evaluate postfix expressions, we are ready to write our program

We will first try to get a simple working version of the algorithm up and running so we can start using it right away

We will add features one at a time, testing them as we go

Postfix Evaluator Input

The first thing is to be clear about the input to our program:

- numbers and operators are called **tokens**
- e.g. -8.886 is a token and **/** is a token
- a postfix expression is thus a sequence of 1 or more tokens
- e.g. the postfix expression **4 3 * 2 +** consists of 5 tokens
- we will require that there be at least one space between each token
- so an expression like **4 3*2+** will cause an error in our evaluator

Reading Tokens

Start by writing the basic code for reading tokens and distinguishing between numbers and operators:

```
#include "error.h"
#include <iostream>
#include <string>
using namespace std;

string token;  // global variable holds both operators and numbers

int main() {
    cout << "Postfix expression evaluator\n";
    while (cin >> token) {
        if (token=="+" || token=="-" || token=="*" || token=="/") {
            cout << "'" << token << "' is an operator\n";
        } else {
             cout << token << " is (perhaps) a number\n";
        }
    }
}
```


Slides based on Problem Solving with C++, 9th Edition, Walter Savitch
and Course Notes on Expression Evaluator by Dr. Toby Donaldson

Adding a Stack

We need to add a stack according to our pseudo-code

Notice the algorithm never stores operators on the stack, only numbers

Since we only ever store numbers on the stack,
we can use a vector of doubles to represent our stack

```
vector<double> stack; // global variable holds only numbers
```

Converting a `string` token to a `double`

Each Token may be either an operator or a number
So need to read tokens as strings
Need to convert strings like **"-3.05"** to a **double -3.05**
Easiest way is to use C++11 function stod():

```
#include <string>

string s = " -5.026  ";
double x = stod(s);
cout << "\"" << s << "\"" << "\n" << x << "\n";
```

This prints:

```
" -5.026  "
-5.026
```

Evaluating the “+” Operator

Now have enough to evaluate postfix expressions that use +

```
string token;           // global variable
vector<double> stack;   // global variable

int main() {
    cout << "Postfix expression evaluator\n";
    while (cin >> token) {
        if (token == "+") {
            // pop the top two elements of the stack
            // the top element is b, and the one under the top is a
            double b = stack.back();
            stack.pop_back();
            double a = stack.back();
            stack.pop_back();
            stack.push_back(a + b);
            // print the top of the stack so we can see the result
            cout << "tos = " << stack.back() << "\n";
        } else {
            stack.push_back(string_to_double(token));
        }
    }
}
```

Sample Test Run for + Operator

Test code available through CourSys: **wk02.1_Slide22**
Sample output: (Keyboard input is Green)

```
Postfix expression evaluator
1 2 +
tos = 3
4 +
tos = 7
3 -1 -2 1 +
tos = -1
<ctrl-d>
RUN FINISHED; exit value 0; real time: 57s; user: 0ms; system: 0ms
```

Refactoring Repeated Code Sequences

Create helper functions for code that gets repeated often

Makes code easier to read, understand, and fix

Notice always call **`stack.back()`** and **`stack.pop_back()`** together

- So it's a good candidate for a helper function to simplify things a bit
- Let's create a helper function to combine both operations into one

```
// remove and return the element of the stack
double pop() {
  double tos = stack.back();
  stack.pop_back();
  return tos;
}
```

For consistency, lets also simplify things with a similar push helper function:

```
// put x on the top of the stack
void push(double x) {
    stack.push_back(x);
}
```

Postfix Evaluator – Simplified with Helper Functions

The code for the + operator is now shorter and easier to read!

```
string token;           // global variable
vector<double> stack;   // global variable

int main() {
    cout << "Postfix expression evaluator\n";
    while (cin >> token) {
        if (token == "+") {
            // pop the top two elements of the stack
            // the top element is b, and the one under the top is a
            double b = pop();
            double a = pop();
            push(a + b);
            // print the top of the stack so we can see the result
            cout << "tos = " << stack.back() << "\n";
        } else {
            push(string_to_double(token));
        }
    }
}
```

Implementing the other Operators: - * /

With “+” code working and simplified, time to code the other operators

```
while (cin >> token) {
  if (token == "+") {
    // pop the top two elements of the stack
    // the top element is b, and the one under the top is a
    double b = pop();
    double a = pop();
    push(a + b);
    // print the top of the stack so we can see the result
    cout << "tos = " << stack.back() << "\n";
  } else if (token == "-") {
    double b = pop();
    double a = pop();
    push(a - b);
    cout << "tos = " << stack.back() << "\n";
  } else if (token == "*") {
    double b = pop();
    double a = pop();
    push(a * b);
    cout << "tos = " << stack.back() << "\n";
  } else if (token == "/") {
    double b = pop();
    double a = pop();
    push(a / b);
    cout << "tos = " << stack.back() << "\n";
  } else {
    push(string_to_double(token));
  }
```

Testing the full version of the Postfix Evaluator


```
Postfix expression evaluator
3 4 *
tos = 12
1 2 + 3 4 + *
tos = 3
tos = 7
tos = 21
<ctrl-d>
RUN FINISHED; exit value 0; real time: 57s; user: 0ms; system: 0ms
```

This version, Postfix_Eval_v3, works well enough.

It took 60 lines of code to implement this postfix expression evaluator

That's pretty good!

Can we do better?

Adding more Features

Lets add more features to our calculator

- right now we print the top of the stack after every operator
- that's useful for debugging but most of the time we usually only care about the final value of our calculation
- so let's replace the top-of-stack printing code with the "=" operator
- the "=" operator immediately prints the top of the stack
- let's add a user prompt "--> "
- let's also add a Q command to quit (hitting <ctrl>-d is not intuitive)

```
Postfix expression evaluator
--> 3 4 * =
tos = 12
--> 1 2 + 3 4 + * =
tos = 21
--> Q
RUN FINISHED; exit value 0; real time: 57s; user: 0ms; system: 0ms
```

Printing the Top of the Stack

To implement =, re-write the main if-else-if statement

- Eliminate the top-of-stack printing associated with each operator
- Add a check for the new = operator, and print the top-of-stack there

```
if (token == "+") {
  double b = pop();
  double a = pop();
  push(a + b);
} else if (token == "-") {
  double b = pop();
  double a = pop();
  push(a - b);
} else if (token == "*") {
  double b = pop();
  double a = pop();
  push(a * b);
} else if (token == "/") {
  double b = pop();
  double a = pop();
  push(a / b);
} else if (token == "=") {
  cout << "tos = " << stack.back() << "\n";
} else {
  push(string_to_double(token));
}
```

Adding the Q operator to exit the while loop

while loops continue to loop until their boolean expression evaluates to **false**
To support Q, this must become **false** when the Q Operator is found

```
while ((cin >> token) && (token != "Q")) {
  if (token == "+") {
     double b = pop();
     double a = pop();
     push(a + b);
  } else if (token == "-") {
     double b = pop();
     double a = pop();
     push(a - b);
  } else if (token == "*") {
     double b = pop();
     double a = pop();
     push(a * b);
  } else if (token == "/") {
     double b = pop();
     double a = pop();
     push(a / b);
  } else if (token == "=") {
     cout << "tos = " << stack.back() << "\n";
  } else {
     push(string_to_double(token));
  }
} cout << "Bye!" << endl;
```

Adding an Input Prompt

A useful feature is to display a text prompt for the user
Need to make three small changes throughout the program:

```
const string prompt = "--> ";

int main() {
    cout << "Postfix expression evaluator\n";
    cout << prompt;
    while ((cin >> token) && (token != "Q")) {
        if (token == "+") {
            double b = pop();
            double a = pop();
            push(a + b);
        }
        // ... code for -, *, and / ...
        } else if (token == "=") {
            cout << "tos = " << stack.back() << "\n";
            cout << prompt;
  } else {
    push(string_to_double(token));
  }
} cout << "Bye!" << endl;
```


Slides based on Problem Solving with C++, 9th Edition, Walter Savitch
and Course Notes on Expression Evaluator by Dr. Toby Donaldson

Summary

Key Things to take away from this presentation:

- While Global variables should be avoided, can be used to simplify complex code
- Stacks are very simple and very power data structures!
- Stacks can be implemented using C++ Vectors
- Helper Functions simplify main() and make the Flow of Control easier to read
- Boolean Expressions control when a while loop breaks out of the loop

References:

1. Walter Savitch, *Problem Solving with C++*. Pearson, 9th Edition, 2014, ISBN 978-0-13-359174-3
2. T.Donaldson, *Parsing Postfix Expressions*, http://www.cs.sfu.ca/CourseCentral/135/tjd/postfix.html
3. B.Fraser*, Install VMWare Player & Creating an Ubuntu VM (Part 1),* https://youtu.be/TXGREvxPbL4
4. B.Fraser, *Configure a VMWare Player VM (Part 2)*, https://youtu.be/WvWsb5fh2fQ
5. B.Fraser, Installing Netbeans for C++ on Ubuntu VM, https://youtu.be/46SDMxtWTSw

Time for Questions