

Stack Applications

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

- 1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions
- 2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix
- 3 Extended Infix to Postfix
infix expressions with brackets

MCS 360 Lecture 15
Introduction to Data Structures
Jan Vershelde, 27 September 2010

Stack Applications

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions

2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix

3 Extended Infix to Postfix
infix expressions with brackets

postfix expressions

Evaluating
Postfix
Expressionsparsing a postfix
expressionevaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

A postfix expression is an arithmetic expression where we *first* write the two operands *and then* the operator.

Some examples with corresponding infix notation:

postfix	infix
9 8 *	9 * 8
3 4 5 + *	(4 + 5) * 3
3 4 + 5 *	(3 + 4) * 5

Advantage of postfix notation: no brackets needed.

reading from strings

Including `<sstream>` we can read from strings:

```
string expression;  
getline(cin, expression, '\\n');  
  
istringstream tokens(expression);  
char next_character;  
  
while(tokens >> next_character)  
    cout << next_character;
```

Observe:

- `while(tokens >> next_character)` means:
as long as extraction of a character succeeds,
- `tokens >> next_character` reads next *nonblank* character, so we naturally skip spaces.

parsing operators

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

```
const string operators = "+-*/";
istringstream arguments(expression);
bool valid = true;
while(arguments >> next_character) {
    if(!isdigit(next_character)) {
        if(operators.find(next_character)
           == string::npos)
        {
            cout << next_character
                 << " is invalid operator" << endl;
            valid = false; break;
        }
    }
    else
        cout << "operator : "
             << next_character << endl;
}
```

getting operands

Evaluating
Postfix
Expressionsparsing a postfix
expressionevaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

```
else
{
    arguments.putback(next_character);
    int value;
    arguments >> value;
    cout << "operand : " << value << endl;
}
```

Observe:

- the `putback()` method *unread*s a character,
- with `arguments >> value` we read any integer.

running parsing expression

Evaluating Postfix Expressions

parsing a postfix expression

evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

Reading from a string twice:

```
$ /tmp/parsing
```

```
Give a postfix expression : 13 44 5 + *
```

```
-> your expression : "13 44 5 + *"
```

```
-> your expression : "13445+*"
```

```
-> parsing expression ...
```

```
operand : 13
```

```
operand : 44
```

```
operand : 5
```

```
operator : +
```

```
operator : *
```

```
$
```

Note: represent -5 as `0 5 -`.

Stack Applications

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions

2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix

3 Extended Infix to Postfix
infix expressions with brackets

Evaluating a Postfix Expression

Evaluating Postfix Expressions

parsing a postfix expression

evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

Given a postfix arithmetical expression, compute its value.

Using a stack, the algorithm evaluates expression as:

```
for every item in expression do
    if (item is operand) then
        push item on the stack
    else
        right = pop from stack
        left = pop from stack
        push (left item right) on the stack
result = pop from stack
```

Our pop is `stack<int>.top()` followed by `stack<int>.pop()`

Tracing the Evaluation

Evaluating Postfix Expressions

parsing a postfix
expression

evaluating postfix
expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with
brackets

```
$ /tmp/test_eval
Give a postfix expression : 3 4 5 + *
-> your expression : "3 4 5 + *"
pushing 3
pushing 4
pushing 5
evaluating +
popping 5
popping 4
pushing 9
evaluating *
popping 9
popping 3
pushing 27
the value of "3 4 5 + *" : 27
$
```

an Object-Oriented Implementation

Evaluating Postfix Expressions

parsing a postfix
expression

evaluating postfix
expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with
brackets

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

int main()
{
    string expression;

    cout << "Give a postfix expression : ";
    getline(cin, expression, '\n');

    Postfix_Evaluator p(expression);

    cout << "the value of \"
        << expression << "\" : \"
        << p.value() << endl;
```

data members of class

Evaluating Postfix Expressions

parsing a postfix expression

evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

```
#ifndef POSTFIX_EVALUATOR_H
#define POSTFIX_EVALUATOR_H
#include <stack>
#include <string>

class Postfix_Evaluator
{
private:

    std::string expression;
    static const std::string operators;
    std::stack<int> operands;
```

We store given expression, the definition of the operators, and the stack of operands.

functions in class

Evaluating
Postfix
Expressionsparsing a postfix
expression
evaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

```
private:
    bool is_operator(char c);
    // returns true if c is an operator

    int eval_operator(char c);
    // returns the value of one operator

public:
    Postfix_Evaluator(std::string s);
    // stores the expression

    int value();
    // returns the value of the expression
```

The functions declared under `private` are auxiliary to the public `value` method.

the file `postfix_evaluator.cpp`

```
const std::string Postfix_Evaluator::operators
    = "+-*/";
bool Postfix_Evaluator::is_operator(char c) {
    return (operators.find(c) != std::string::npos);
}
int Postfix_Evaluator::eval_operator(char c)
{
    int right = this->operands.top();
    this->operands.pop();
    int left = this->operands.top();
    this->operands.pop();
    switch(c)
    {
        case '+': return left + right;
        case '-': return left - right;
        case '*': return left * right;
        case '/': return left / right;
    }
}
```

Evaluating
Postfix
Expressions

parsing a postfix
expression

evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

the `value()` method

```
int Postfix_Evaluator::value()
{
    std::istringstream tokens(this->expression);
    char c;
    while(tokens >> c)
        if(is_operator(c))
        {
            int v = eval_operator(c);
            this->operands.push(v);
        }
        else
        {
            tokens.putback(c);
            int operand;
            tokens >> operand;
            this->operands.push(operand);
        }
    return operands.top();
}
```

Stack Applications

Evaluating Postfix Expressions

parsing a postfix expression
evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions

2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix

3 Extended Infix to Postfix
infix expressions with brackets

from infix to postfix

Evaluating
Postfix
Expressionsparsing a postfix
expression
evaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

Given a sum of integers, operators + and -, convert:

```
$ /tmp/convertimg
```

```
Give an infix sum : 234 - 88 + 921 - 8
```

```
-> your expression : "234 - 88 + 921 - 8"
```

```
postfix notation of "234 - 88 + 921 - 8" \
```

```
is "234 88 - 921 + 8 -"
```

```
$
```

Algorithm stores the previous operator.

For every item in the infix expression:

- if item is operand, write to result;
- if item is operator, write the previous operator and store item as the previous operator.

At end, write the previous operator to result.

writing to string

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
    string expression;

    cout << "Give an infix sum : ";
    getline(cin, expression, '\n');

    istringstream arguments(expression);
    char next_character;
    char previous_op = ' ';
    ostringstream postfix_expression;
```

converting expression

Evaluating
Postfix
Expressionsparsing a postfix
expression
evaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

```
while(arguments >> next_character)
{
    if(!isdigit(next_character)) // assume + or -
    {
        if(previous_op != ' ')
            postfix_expression << previous_op << " ";
        previous_op = next_character;
    }
    else
    {
        arguments.putback(next_character);
        int value;
        arguments >> value;
        postfix_expression << value << " ";
    }
}
postfix_expression << previous_op;
```

Stack Applications

Evaluating Postfix Expressions

parsing a postfix expression
evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions

2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix

3 Extended Infix to Postfix
infix expressions with brackets

Converting Infix to Postfix

Evaluating Postfix Expressions

parsing a postfix expression
evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

Consider infix expressions with operators $+$, $-$, $*$, and $/$, without brackets.

Comparing " $2 + 3 / 4$ " with " $2 / 3 + 4$ ",
or equivalently " $2 3 4 / +$ " with " $2 3 / 4 +$ "
we see that precedence order of operators matters.

We maintain a stack of operators.

Comparing the current operator with the top of the stack, we pop operators as long as their precedence is higher or equal than the precedence of the current operator.

running the conversion

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

```
$ /tmp/test_convert
```

```
Give an infix expression : 2 + 3/4
```

```
pushing +
```

```
pushing /
```

```
popping /
```

```
popping +
```

```
the postfix notation of "2 + 3/4" : 2 3 4 / +
```

```
$ /tmp/test_convert
```

```
Give an infix expression : 2/3 + 4
```

```
pushing /
```

```
popping /
```

```
pushing +
```

```
popping +
```

```
the postfix notation of "2/3 + 4" : 2 3 / 4 +
```

The Conversion Algorithm

Given expression in infix notation.

for every item in expression do

 if (item is operator)

 then process the operator;

 else write item to result;

pop all operators from stack and write.

To process the current operator:

 if (the stack is empty) then

 push current operator on the stack;

 else if (current operator > operator on top) then

 push current operator on the stack;

 else do

 pop operator on top and write;

 break if (stack is empty);

 while (current operator \leq operator on top);

 push current operator on the stack.

an Object-Oriented Implementation

Evaluating Postfix Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with
brackets

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

int main()
{
    string expression;

    cout << "Give an infix expression : ";
    getline(cin,expression,'\n');

    Infix_to_Postfix e(expression);

    cout << "the postfix notation of \"
        << expression << "\" : \"
        << e.convert() << endl;
```

data members

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

```
#ifndef INFIX_TO_POSTFIX_H
#define INFIX_TO_POSTFIX_H
#include <stack>
#include <sstream>
#include <string>

class Infix_to_Postfix
{
private:
    std::string expression;
    static const std::string operators;
    std::stack<char> operator_stack;
    std::ostringstream postfix_expression;
};
```

To share the result `postfix_expression` between various methods, we make it a data attribute.

methods of the class

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

```
private:
    bool is_operator(char c);
    // returns true if c is an operator
    int precedence_operator(char c);
    // returns precedence of operator c
    void process_operator(char c);
    // processes one operator

public:
    Infix_to_Postfix(std::string s);
    // stores the expression
    std::string convert();
    // returns the equivalent postfix expression
```

the file `infix_to_postfix.cpp`Evaluating
Postfix
Expressionsparsing a postfix
expression
evaluating postfix
expressionsConverting
Infix to Postfix
Expressionsconverting infix sums
from infix to postfixExtended Infix
to Postfixinfix expressions with
brackets

```
const std::string Infix_to_Postfix::operators
    = "+-*/";

bool Infix_to_Postfix::is_operator(char c)
{
    return (operators.find(c) != std::string::npos);
}

int Infix_to_Postfix::precedence_operator(char c)
{
    if(c == '+' || c == '-')
        return 1;
    else
        return 2;
}
```

the convert() method

```

std::string Infix_to_Postfix::convert() {
    std::istringstream tokens(this->expression);
    char c;
    while(tokens >> c)
        if(is_operator(c))
            process_operator(c);
        else {
            tokens.putback(c);
            int operand;
            tokens >> operand;
            this->postfix_expression << operand << " ";
        }
    while(!this->operator_stack.empty()) {
        char c = this->operator_stack.top();
        this->postfix_expression << c << " ";
        this->operator_stack.pop();
    }
    return this->postfix_expression.str();
}

```

process_operator

```

void Infix_to_Postfix::process_operator(char c) {
    if(this->operator_stack.empty())
        this->operator_stack.push(c);
    else {
        char top_op = this->operator_stack.top();
        if(precedence_operator(c)
            > precedence_operator(top_op))
            this->operator_stack.push(c);
        else {
            do {
                this->postfix_expression << top_op << " ";
                this->operator_stack.pop();
                if(this->operator_stack.empty()) break;
                top_op = this->operator_stack.top();
            } while(precedence_operator(c)
                    <= precedence_operator(top_op));
            this->operator_stack.push(c);
        }
    }
}

```

Evaluating
Postfix
Expressions

parsing a postfix
expression
evaluating postfix
expressions

Converting
Infix to Postfix
Expressions

converting infix sums
from infix to postfix

Extended Infix
to Postfix

infix expressions with
brackets

Stack Applications

Evaluating Postfix Expressions

parsing a postfix expression
evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums
from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

- 1 Evaluating Postfix Expressions
parsing a postfix expression
evaluating postfix expressions
- 2 Converting Infix to Postfix Expressions
converting infix sums
from infix to postfix
- 3 Extended Infix to Postfix
infix expressions with brackets

Infix with Brackets to Postfix

Evaluating Postfix Expressions

parsing a postfix expression
evaluating postfix expressions

Converting Infix to Postfix Expressions

converting infix sums from infix to postfix

Extended Infix to Postfix

infix expressions with brackets

To convert "3 * (4 + 5)" to "3 4 5 + *"
we view brackets (and) as operators.

To process a bracket operator:

- an opening bracket (is pushed on the stack
- at a closing bracket), we pop all operators up to and including the matching opening bracket.

Modifications to `Infix_to_Postfix`:

- 1 Add (and) to static const string operators.
- 2 Define precedence of the brackets as zero.
- 3 Extend `process_operator` to test for) and (.

Summary + Assignments

Expression manipulations use stack, we covered §5.4.

Assignments:

- 1 Define exception `Stack_Empty` to be thrown when postfix evaluator does `top` or `pop` on empty stack. Modify the class `Postfix_Evaluator` and test it on invalid postfix expressions.
- 2 Adjust one of our own stack applications of L-14 with a `to_string()` method that sends the content of the stack to a string. Use this `to_string()` to print the evolution of the stack in `Postfix_Evaluator`.
- 3 Extend the `Infix_to_Postfix` methods so they throw exceptions when the stack turns out empty when an invalid infix expression is given. Provide a handler (`try - catch` block) in the test program.