Ch 7
Stacks

- Introduction to Stacks and the STL Stack
- Stack Applications
- Implementations of the Stack Class
- More Complex Stack Applications

Introduction to Stacks and the STL Stack Class

Basic Definitions and Operations

Basic Definitions

- A **type** is a collection of values.
- A **data type** is a type together with a collection of operations to manipulate this type.
- A **data item** is a datum drawn from a type's members.
- Data items may be **simple** or **aggregate**.
- An abstract data type (**ADT**) defines a data type in terms of its interface in a formal language. The implementation is not specified.
- A **data structure** defines the physical forms and algorithms that implement an ADT.

Stack Definitions

- A **stack** is a **data type** (not a data structure) consisting of ordered **data items** such that they may be inserted and removed at and from only one position in the order, called the **top**.
- Stacks obey last-in/first-out (LIFO) rules. Data items are removed from a stack in the reverse order of their insertion.
- Stacks, therefore, are containers or **list** data types with specific additional properties.

The **stack** STL Class

The stack class is formally declared as:

```
template <class T, class Container = deque<T>>
Class stack { /* ... */ };
```

Its constructor has the form:

```
explicit stack(const Container& cnt = Container());
```

Mutator functions:

```
void pop();
void (push(const T& val);
```

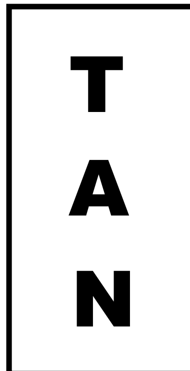
Accessor functions:

```
bool empty() const;
size_type& size() const;
value_type& top();
const value_type& top() const;
```

Example: Reversing a Word

```
while (there are more characters)
  Read a character.
  Push the character onto a stack.
while (the stack is not empty)
  Write out the top character.
  Pop the stack.
```

Stack



Input

Output

Stack Applications

Balancing Parentheses & Evaluating Expressions

Example: Balanced Parentheses

```
failed = false
while (there are more characters)
   if (next character == '(')
      push next character
   if (next character == ')' and stack is not empty)
      pop
   else if (next character == ')' and stack is empty)
      failed = true
return stack is empty && not failed
```

Try: (( ) ( ))

parens.cxx...

```
#include <cstdlib>
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isBalanced(const string& exp);

int main( ) {
  string user_input;
  cout << "Type a string with some parentheses:\n";
  getline(cin, user_input);

  if (isBalanced(user_input))
    cout << "Those parentheses are balanced.\n";
  else
    cout << "Those parentheses are not balanced.\n";

  cout << "That ends this balancing act.\n";
  return EXIT_SUCCESS;
}
```

…parens.cxx

```
bool isBalanced(const string& exp) {
  const char LEFT = '(';
  const char RIGHT = ')';
  stack<char> store;
  string::size_type i;
  char next;
  bool failed = false;

  for (i = 0; !failed && (i < exp.length( )); ++i) {
    next = exp[i];
    if (next == LEFT)
      store.push(next);
    else if ((next == RIGHT) && (!store.empty( )))
      store.pop( );
    else if ((next == RIGHT) && (store.empty( )))
      failed = true; }

  return (store.empty( ) && !failed);
}
```

Example: Calculator

```
while (there are more characters)
   if (next item is a number)
      read and push onto the numbers stack
   else if (next item is an operation)
      read and push onto the operations stack
   else if (next item is ')')
      pop two numbers and one operation
      apply the operation
      push the result onto the numbers stack
   else if (next item is "(" or blank)
      read and ignore
return the top of the numbers stack
```

Try: `(((6 + 9) / 3) * (6 - 4))`

calc.cxx...

```
#include <cctype>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <stack>
using namespace std;

double readAndEval(istream& ins);
void evalTops(stack<double>& nums, stack<char>& ops);

int main( ) {
  double answer;

  cout << "Type a fully parenthesized arithmetic expression:" << endl;
  answer = readAndEval(cin);
  cout << "That evaluates to " << answer << endl;

  return EXIT_SUCCESS;
}
```

...calc.cxx...

```
double readAndEval(istream& ins) {
  const char POINT = '.';
  const char RIGHT = ')';
  stack<double> nums;
  stack<char> ops;
  double number;
  char symbol;

  while (ins && ins.peek( ) != '\n') {
    if (isdigit(ins.peek( )) || (ins.peek( ) == POINT)) {
      ins >> number;
      nums.push(number); }
    else if (strchr("+-*/", ins.peek( )) != NULL) {
      ins >> symbol;
      ops.push(symbol); }
    else if (ins.peek( ) == RIGHT) {
      ins.ignore( );
      evalTops(nums, ops); }
    else
      ins.ignore( ); }
  return nums.top( );
}
```

...calc.cxx

```
void evalTops(stack<double>& nums, stack<char>& ops) {
  double num2 = nums.top( ); nums.pop( );
  double num1 = nums.top( ); nums.pop( );

  switch (ops.top( )) {
    case '+':
      nums.push(num1 + num2);
      break;
    case '-':
      nums.push(num1 - num2);
      break;
    case '*':
      nums.push(num1 * num2);
      break;
    case '/':
      nums.push(num1 / num2);
      break; }
  ops.pop( );
}
```

Implementations of the Stack Class

Array and Linked List Implementations

stack1.h

```
template <class Item>
class stack {
  public:
    typedef std::size_t size_type;
    typedef Item value_type;
    static const size_type CAPACITY = 30;

    stack( ) { used = 0; }

    void push(const Item& entry);
    void pop( );

    bool empty( ) const { return (used == 0); }
    size_type size( ) const { return used; }
    Item top( ) const;
  private:
    Item data[CAPACITY];
    size_type used;
};
```

stack1.template

```
template <class Item>
const typename stack<Item>::size_type stack<Item>::CAPACITY;

template <class Item>
void stack<Item>::push(const Item& entry) {
  assert(size( ) < CAPACITY);
  data[used] = entry;
  ++used;
}

template <class Item>
void stack<Item>::pop( ) {
  assert(!empty( ));
  --used;
}

template <class Item>
Item stack<Item>::top( ) const {
  assert(!empty( ));
  return data[used -1];
}
```

stack2.h

```
#include <cstdlib>
#include "node2.h"

template <class Item>
class stack {
  public:
    typedef std::size_t size_type;
    typedef Item value_type;

    stack( ) { top_ptr = NULL; }

    stack(const stack& source);
    void operator =(const stack& source);
    ~stack( ) { list_clear(top_ptr); }

    void push(const Item& entry);
    void pop( );

    size_type size( ) const { return list_length(top_ptr); }
    bool empty( ) const { return (top_ptr == NULL); }
    Item top( ) const;
  private:
    node<Item> *top_ptr;
}
```

stack2.template…

```
template <class Item>
stack<Item>::stack(const stack<Item>& source) {
  node<Item> *tail_ptr;
  list_copy(source.top_ptr, top_ptr, tail_ptr);
}

template <class Item>
void stack<Item>::operator =(const stack<Item>& source) {
  node<Item> *tail_ptr;
  if (this == &source)
    return ;
  list_clear(top_ptr);
  list_copy(source.top_ptr, top_ptr, tail_ptr);
}
```

…stack2.template

```
template <class Item>
void stack<Item>::push(const Item& entry) {
  list_head_insert(top_ptr, entry);
}

template <class Item>
void stack<Item>::pop( ) {
  assert(!empty( ));
  list_head_remove(top_ptr);
}

template <class Item>
Item stack<Item>::top( ) const {
  assert(!empty( ));
  return top_ptr->data( );
}
```

More Complex Stack Applications

General Postfix and Infix Expression Evaluation

Example: Postfix Evaluation

```
do
   if (next item is a number)
      read and push
   else
      read the operator
      pop twice
      apply the operator
      push
while (there are more items)
return the top of the stack
```

Try: **5 3 2 * + 4 - 5 +**

Example: Convert Infix to Postfix

```
do
   if (next item is '(')
      read and push
   else if (next item is an operand)
      read and write
   else if (next item is an operator)
      read and push
   else
      read and discard the ')'
      pop and write the operator
      pop and discard the '('
while (there is more input)
```

Try: `(((2 * (A - B)) + 3) + C)`

Example: General Conversion

```
do
   if (next item is a '(')
      read and push
   else if (next item is an
              operand)
      read and write
   else if (next item is an
              operator)
      execute procedure operator
      read and push
   else
      execute procedure rParen
while (not done)
pop and write remaining operators
```

```
operator:
while (stack not empty
  and top is not '('
  and top is not lower
    than next item)
      pop and write
```

```
rParen:
read and discard the ')'
while (top is not '(')
  pop and write
pop the '('
```

Try: **3 * X + (Y - 12) - Z**

CIS 11: Data Structures and Algorithms

Project F

Complete Project 10 from Chapter 7 of the text. Directions are on page 387.

- Use the program available here as your starting point instead of the one from the book. The primary difference is that the one you'll modify reads it's expressions to evaluate from a text file named `expressions.dat`.
- You may use the `expression class`, found in expression.h, to hold the intermediate postfix expression if you wish.
- Test your program thouroughly with your own test file. You will not submit that file.

When you're satisfied with the program, submit your work via the CATE form for Project F.

calc.cxx…

```
#include <cctype>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <fstream>
#include <stack>
#include <string>

using namespace std;

void open_files(ifstream&, ifstream&);
double read_and_evaluate(istream& ins);
void evaluate_stack_tops(stack<double>& numbers,
                         stack<char>& operations);
```

…calc.cxx…

```
int main( ) {
  double answer;
  string s;
  ifstream in, exp;

  open_files(in, exp);
  while (in) {
    answer = read_and_evaluate(in);
    getline(exp, s);
    cout << s << " = " << answer << endl; }
  in.close();
  exp.close();
  return EXIT_SUCCESS;
}
```

…calc.cxx…

```
void open_files(ifstream& i, ifstream& e) {
  i.open("expressions.dat");
  e.open("expressions.dat");
  if (i.fail()) {
    cout << "Attempt to open expressions.dat on \"in\" failed."
         << endl;
    exit(1); }
  if (e.fail()) {
    cout << "Attempt to open expressions.dat on \"exp\" failed."
         << endl;
    exit(1); }
}
```

...calc.cxx...

```
double read_and_evaluate(istream& ins) {
  const char DECIMAL = '.';
  const char RIGHT_PARENTHESIS = ')';
  stack<double> numbers;
  stack<char> operations;
  double number;
  char symbol;

  while (ins && ins.peek( ) != '\n') {
    if (isdigit(ins.peek( )) || (ins.peek( ) == DECIMAL)) {
      ins >> number;
      numbers.push(number); }
    else if (strchr("+-*/", ins.peek( )) != NULL) {
      ins >> symbol;
      operations.push(symbol); }
    else if (ins.peek( ) == RIGHT_PARENTHESIS) {
      ins.ignore( );
      evaluate_stack_tops(numbers, operations); }
    else {
      ins.ignore(); } }
  ins.ignore();
  return numbers.top( );
}
```

…calc.cxx

```
void evaluate_stack_tops(stack<double>& numbers,
                         stack<char>& operations) {
  double operand1, operand2;

  operand2 = numbers.top( );
  numbers.pop( );
  operand1 = numbers.top( );
  numbers.pop( );
  switch (operations.top( )) {
    case '+':
      numbers.push(operand1 + operand2);
      break;
    case '-':
      numbers.push(operand1 - operand2);
      break;
    case '*':
      numbers.push(operand1 * operand2);
      break;
    case '/':
      numbers.push(operand1 / operand2);
      break; }
  operations.pop( );
}
```

`expression` Constructors

```
expression() { }
expression(std::istream&);
expression(std::string);
```

expression Tests

```
bool nextIsNumber();
bool nextIsOperator();
bool nextIsRight();
bool nextIsLeft();
bool hasNext();
```

expression Accessors

```
double getNumber() throw (std::range_error);
char getOperator() throw (std::range_error);
char getParenthesis() throw (std::range_error);
std::string nextToString();
```

expression Mutators

```
void putNumber(double);
void putOperator(char)
  throw (std::invalid_argument);
void putParenthesis(char)
  throw (std::invalid_argument);
```

expression Output

```
void putExpression(std::ostream&);
void putExpression(std::string&);
```