Stacks and Queues

Stacks

Stacks are used for:

- Solving mazes
- Undo in your word processor
- Evaluating math expression
- Tracking hwere to return from C++ function calls

There is a stack hardwired into every CPU!

A stack is an ADT that holds a collection of items where the elements are always added to one end. The last item pushed onto the stack is the first item to be removed.

Stack operations:

- put something on top of the stack: PUSH
- remove the top item: POP
- look at the top item
- check to see if the stack is empty

```
class Stack

{

    public:

        Stack();

        void push(int i);

        int pop();

        bool is_empty(void);

        int peek_top();

    private:

        ...

}
```

Implementing the stack:

```
const int size = 100;

class Stack
```

```
{
    public:
        Stack()
        {
            m_top = 0;
        }
        //when we push, we store the new item in m_stack[m_top]
        //and post-increment our m_top variable
        void push(int val)
        {
            if(m_top >= SIZE)
                return;
            m_stack[m_top] = val;
            m_top += 1;
        }
        //when we pop, we pre-decrement our m_top variable
        //and return the item in m_stack[m_top]
        int pop()
        {
            if(m_top == 0)
                return -1;
            m_top -= 1;
            return m_stack[m_top];
        }
        ...
    private:
        int m_stack[SIZE];
        int m_top;
}
```

Stacks are so popular that the C++ people wrote one for you!
To use the stack, `#include<stack>`

```
#include <iostream>

#include <stack>

int main()

{

    std::stack<int> istack;

    istack.push(10);

    istack.push(20);

    cout << istack.top();

    istack.pop();

    if(istack.empty() == false)

        cout << istack.size();

}
```

From this code snippet, you can see that the methods of the C++ stack from the STL are:

- `push()` : adds an item onto the top of the stack
- `pop()` : throws away the top item from the stack
- `top()` : returns the top item on the stack
- `empty()` : returns true if the stack is empty
- `size()` : returns the size of the stack

Infix to Postfix Conversion example:
(3 + 5) * (4 + 3 / 2) - 5 ------> 3 5 + 4 3 2 / + * 5
Input: infix string
Output: postfix string
Private data: a stack
Algorithm:

1. Begin at left-most Infix token
2. If it's a #, apend it to the end of postfix string followed by a space
3. If it's a "(", push it onto the stack
4. If it's an operator and the stack is empty, push the operator on the stack.
5. If it's an operator and the stack is not empty, pop all operators with greater or equal precedence off the stack and append them on the postfix string. Stop when you reach an operator with lower precedence or a "(", and push the new operator on the stack.
6. If you encounter a ")", pop operators off the stack and append them onto the postfix string until you pop a matching "(".
7. Advance to next token and go back to #2.
8. When all infix tokens are gone, pop each operator and append it to the postfix string.

```
#include <iostream>

#include <string>
```

```cpp
#include <cctype>

#include <stack>

#include <cmath>

using namespace std;


//returns 0 if op1 has greater or equal precedence than op2

//returns 1 if op1 has less precedence than op2

int compare(char op1, char op2)

{

    if(op1 == '+')

    {

        if(op2 == '-' || op2 == '+')

            return 0;

        return 1;

    }

    if(op1 == '-')

    {

        if(op2 == '+' || op2 == '-')

            return 0;

        return 1;

    }

    if(op1 == '*')

    {

        if(op2 == '/' || op2 == '+' || op2 == '-' || op2 == '*')

            return 0;

        return 1;

    }

    if(op1 == '/')

    {

        if(op2 == '*' || op2 == '+' || op2 == '-' || op2 == '/')

            return 0;

        return 1;
    }

    return -1;

}



//convert infix string to postfix string

string convert(string infix)

{

    stack<string> s_stack;

    string postfix = "";

    //1. Begin at left-most Infix token

    for(int i = 0; i < infix.length(); i++)

    {

        //if it's a number, append it to end of postfix string followed by a space

        if(isdigit(infix[i]))

        {


            while(i < infix.length() - 1 && isdigit(infix[i + 1]))

            {

                postfix += infix.substr(i, 1);

                i++;

            }


            postfix += infix.substr(i, 1) + " ";

        }

        //if it's a (, push it onto the stack

        if(infix.substr(i, 1) == "(")

        {

            s_stack.push("(");

        }

        //if it's an operator

        if(infix[i] == '+' || infix[i] == '-' || infix[i] == '*' || infix[i] == '/')

        {
            //and the stack is empty, push the operator on the stack
            if(s_stack.empty())
            {
                s_stack.push(infix.substr(i, 1));
            }
            else
            {
                //if the stack is not empty
                //pop all operators with greater or equal precedence off the stack
                //and append them on the postfix string
                while(!s_stack.empty() && s_stack.top() != "(" && compare(s_stack.top()[0], infi
x[i]) == 0)
                {
                    postfix += s_stack.top() + " ";
                    s_stack.pop();
                }
                //stop when you reach an operator with lower precedence of a (
                //push the new operator on the stack
                s_stack.push(infix.substr(i, 1));
            }
        }
        //if it's a ), pop operators off the stack and append them onto the postfix string until
 you pop a matching (
        if(infix.substr(i, 1) == ")")
        {
            //cout << ")" << endl;
            string s = s_stack.top();
            while(s != "(")
            {
                postfix += s_stack.top() + " ";
                s_stack.pop();
                s = s_stack.top();
            }
            s_stack.pop();
        }
    }
    //when all infix tokens are gone, pop each operator and append it to the postfix string
    while(!s_stack.empty())
    {
        postfix += s_stack.top() + " ";
        s_stack.pop();
    }
    return postfix;
}

int helper(char num)
{
    switch(num)
    {
        case 0: return 0;
        case 1: return 1;
        case 2: return 2;
        case 3: return 3;
        case 4: return 4;
        case 5: return 5;
        case 6: return 6;
        case 7: return 7;
        case 8: return 8;
        case 9: return 9;
    }
    return -1;
}

double stringToInt(string num)
{
    double j = num.length() - 1;

    double result = 0;

    for(int i = 0; i < num.length(); i++)

    {

        result += helper(num[i]) * pow(10, j);

        j--;

    }

    return result;

}


int main(int argc, const char * argv[]) {


    string infix;

    string postfix;

    string post_fix;

    string result;

    cout << "Enter an infix string: ";

    getline(cin, infix);

    postfix = convert(infix);

    cout << postfix << endl;


    return 0;

}
```

Queues

Queues are used for:

- Optimal route navigation
- Streaming video buffering
- Searching through mazes
- Tracking calls in call centers

FIRST IN, FIRST OUT data structure

Every queue has a front and a rear. You enqueue items at the rear and dequeue from the front.

The queue interface:

- `enqueue(int a)` : inserts an item on the rear of the queue
- `int dequeue()` : removes and returns the top item from the front of the queue
- `bool isEmpty()` : determines if the queue is empty
- `int size()` : determines the number of items in the queue
- `int getFront()` : gets the value of the top item on the queue without removing it like dequeue

The queue class in the Standard Template Library:

```
#include <iostream>

#include <queue>


int main()

{

    std::queue<int> iqueue;

    iqueue.push(10);

    iqueue.push(20);

    cout << iqueue.front();

    iqueue.pop();

    if(iqueue.empty() == false)

        cout << iqueue.size();

}
```

As we can see, the methods of the STL queue are similar to the STL stack:

- `push()`
- `pop()`
- `front()`
- `empty()`
- `size()`