3.2 The Stack Abstract Data Type (Stack ADT or Abstract Stack)

We will look at the stack as our first abstract data type and we will see two different implementations: one using a singly linked list, the other a one-ended array.

3.2.1 Description

An abstract stack is an abstract data type that stores objects in an explicitly defined linear order. The insertion and erase operations are significantly restricted:

1. Objects are always *pushed* onto the stack,
2. The *top* of the stack is the object that was most recently pushed onto the stack, and
3. When an object is *popped* from the stack, it removes and returns the current top of the stack.

Visually, we have these operations as shown in Figure 1.

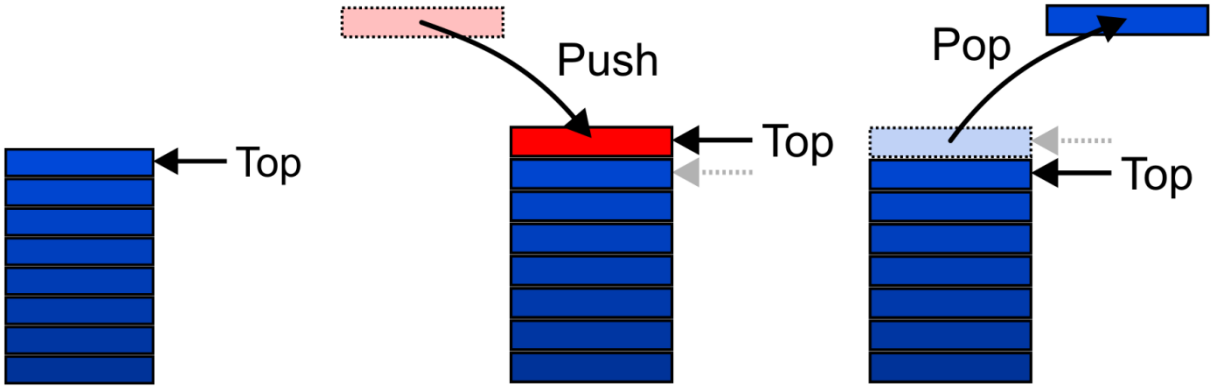

Figure 1. The top, push, and pop operations on a stack.

It is an undefined operation to call top or pop when the stack is empty.

This is also called a *last-in—first-out* (LIFO) behaviour.

This is an extremely simple ADT. The objective in engineering is to try to determine when is to formulate a problem so that it uses a stack.

Stacks are when parsing code, when dealing with function calls, tracking undo and redo operations in most applications, evaluating expressions in reverse Polish, and in formulating assembly language instructions.

Stacks are even used in problem solving: you start working on a large problem, but to solve the larger problem, you have to solve a smaller sub-problem. You put the larger problem on a stack and start focusing on the sub-problem. In solving the sub-problem, you may find you have to solve a sub-sub-problem, so you put the sub-problem on the stack and work on the sub-sub-problem. When you solve a problem, you use its solution and continue working on the problem that is currently on the top of the stack. [Knuth]

3.2.2 Applications

The stack data structure is exceptionally straight-forward. Due to its simplicity, the goal in any engineering problem is to attempt to formulate a solution that makes use of a stack.

Examples of applications include:

1. Parsing code, including:
    a. HTML and XML, and
    b. Matching parentheses in C++,
2. Allocating memory for function calls,
3. Evaluating reverse-Polish expressions,
4. Tracking undo and redo operations in applications (going forward and back in a web browser),
5. Assembly language, and
6. Robert's Rules of Order.

3.2.3 Implementation

The optimal asymptotic behaviour of any function is $\Theta(1)$: the run time is independent of the number of objects being stored in the container. We will attempt to achieve this optimal bound.

3.2.3.1 Singly Linked List Implementation

The class definition of the `Single_list` class implemented in Project 1 is shown here:

```
template <typename Type>
class Single_list {
    public:
        Single_list();
        ~Single_list();

        int size() const;
        bool empty() const;
        Type front() const;
        Type back() const;
        Single_node<Type> *head() const;
        Single_node<Type> *tail() const;
        int count( const Type & ) const;

        void push_front( const Type & );
        void push_back( const Type & );
        Type pop_front();
        int erase( const Type & );
};
```

A singly linked list only allows both Θ(1) insertions and Θ(1) removals from the front of the linked list. We will therefore restrict ourselves to the functions

1. `bool empty() const` Return true if the linked list is empty, and false otherwise.
2. `Type front() const` Return the object at the front of the linked list.
3. `void push_front( Type const &obj )`
   Push the argument `obj` onto the front of the linked list.
4. `Type pop_front()` Pop the object off the front of the linked list.

The class will contain a `Single_list` member variable and, in each case, we will call an appropriate member function on that list.

```
template <typename Type>
class Stack {
    private:
        Single_list<Type> list;

    public:
        bool empty() const;
        Type top() const;

        void push( const Type & );
        Type pop();
};

template <typename Type>
bool Stack<Type>::empty() const {
    return list.empty();
}

template <typename Type>
void Stack<Type>::push( const Type &obj ) {
    list.push_front( obj );
}

template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw underflow();
    }

    return list.front();
}
```

```
template <typename Type>
Type Stack<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }
    return list.pop_front();
}
```

Because the `Stack` class essentially just calls member functions from the `Single_list` class, we call the `Stack` class a *wrapper* class.

3.2.3.2 One-ended Array Implementation

A one-ended array only allows Θ(1) insertions and erases at the back and thus we will restrict our operations to that part of the array.

```
#include <algorithm>

template <typename Type>
class Stack {
    private:
        int stack_size;
        int array_capacity;
        Type *array;

    public:
        Stack( int = 10 );
        ~Stack();
        bool empty() const;
        Type top() const;
        void push( const Type & );
        Type pop();
};

template <typename Type>
Stack<Type>::Stack( int n ):
stack_size( 0 ),
array_capacity( std::max(0, n) ),
array( new Type[array_capacity] )
{
    // Empty constructor
}
```

```
template <typename Type>
Stack<Type>::~Stack() {
    delete [] array;
}

template <typename Type>
bool Stack<Type>::empty() const {
    return ( stack_size == 0 );
}


template <typename Type>
Type Stack<Type>::top() const {
    if ( empty() ) {
        throw underflow();
    }

    return array[stack_size - 1];
}

template <typename Type>
Type Stack<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    --stack_size;
    return array[stack_size];
}

template <typename Type>
void Stack<Type>::push( const Type &obj ) {
    if ( stack_size == array_capacity ) {
        throw overflow();  // Best solution?????
    }

    array[stack_size] = obj;
    ++stack_size;
}
```

The description of the Stack ADT does not state that the stack has a maximum size. Consequently, when the array is full, it is an issue with the implementation and not with the ADT. The above solution throws an exception if the array is full; however, there are numerous possible solutions:

1. We could increase the size of the array,
2. Throw an exception,
3. Ignore the object being pushed onto the stack,
4. Replace the current top of the stack,
5. Drop off the bottom of the stack and push the object onto the top, or
6. Put the process pushing the object onto the stack to sleep until another process removes the top of the stack.

The first is the only solution that produces a *faithful* implementation of the Stack ADT. The last will only be possible in a shared environment. In cases 2, 3, 4, and 5, it would probably be a good idea to provide the function `bool Stack<Type>::full() const;` so that the user can query the state of the stack.

3.2.4 Increasing the Capacity of the Array

If an array is full and we wish to increase its capacity, should we increase the current capacity by a constant, or should we multiply the capacity by a constant (for example, double the capacity)?

Both of these will either adversely affect memory use or the run time. Choosing the appropriate solution will depend on the requirements.

For the array-based implementation of a stack, we could add the member function

```
// Double the capacity of the array
template <typename Type>
void Stack<Type>::double_capacity() {
    Type *tmp_array = new Type[2*array_capacity];

    for ( int i = 0; i < array_capacity; ++i ) {
        tmp_array[i] = array[i];
    }

    delete [] array;
    array = tmp_array;
    array_capacity *= 2;
}
```

To determine the effects of each, we must first introduce a new topic: *amortized run times*.

> If a function requires $\Theta(T(n))$ time to evaluate n operations, we will say that the function has an *amortized run time* of $\Theta(T(n)/n)$ for each operation.

If the run time of a function is always the same, the amortized run time will be the same as the run time. Amortized run times become relevant when the same function will have different run times base on different situations.

For example, pushing something onto a stack when the array is not full is a $\Theta(1)$ operation, but pushing something onto a stack when the array is full is a $\Theta(n)$ operation. If we perform a large number of pushes onto a stack, will the amortized run time be $\Theta(1)$, $\Theta(n)$, or something in between?

3.2.4.1 Increasing the Capacity of the Array by 1

If we increase the array capacity by 1 each time the array is full, when pushing the k^{th} object onto the stack, we will require $k - 1$ copies. The effect of this is shown in Figure 2.

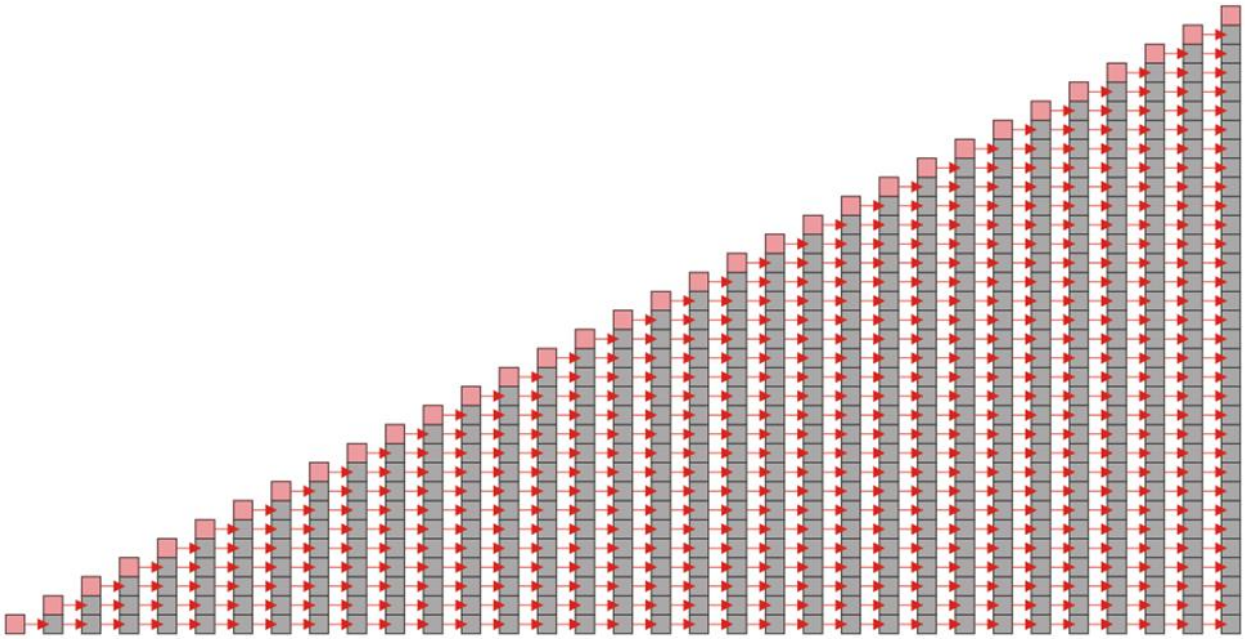

Figure 2. The number of copies required to insert *n* objects when increasing the capacity by one.

Therefore, in pushing *n* objects onto the stack, the total number of copies will be

$$\begin{aligned}\sum_{k=1}^{n}(k-1) &= \left(\sum_{k=1}^{n} k\right) - n \\ &= \frac{n(n+1)}{2} - n \\ &= \frac{n(n-1)}{2} = \Theta\left(n^2\right).\end{aligned}$$

Therefore, the amortized run time will be $\Theta\left(\frac{n^2}{n}\right) = \Theta(n)$. There is never any unused memory.

3.2.4.2 Doubling the Capacity of the Array

Suppose an array has a capacity of 8 and a push operation is performed. Eight copies are made, the capacity is doubled to 16, and the next 7 pushes can each be performed in $\Theta(1)$ time. At this point, the array will be full, so the next push will require 16 copies and the new capacity will be 32. At this point, however, then next 15 pushes can be performed all in $\Theta(1)$ time. Therefore, when inserting *n* objects, we will only have to increase the size of the array with the 2nd, 3rd, 5th, 9th, 17th, 33rd, 65th, *etc*. insertions requiring 1, 2, 4, 8, 16, 32, 64, *etc*. copies, respectively. This is demonstrated in Figure 3 where most insertions do not require changes to the capacity of the array.

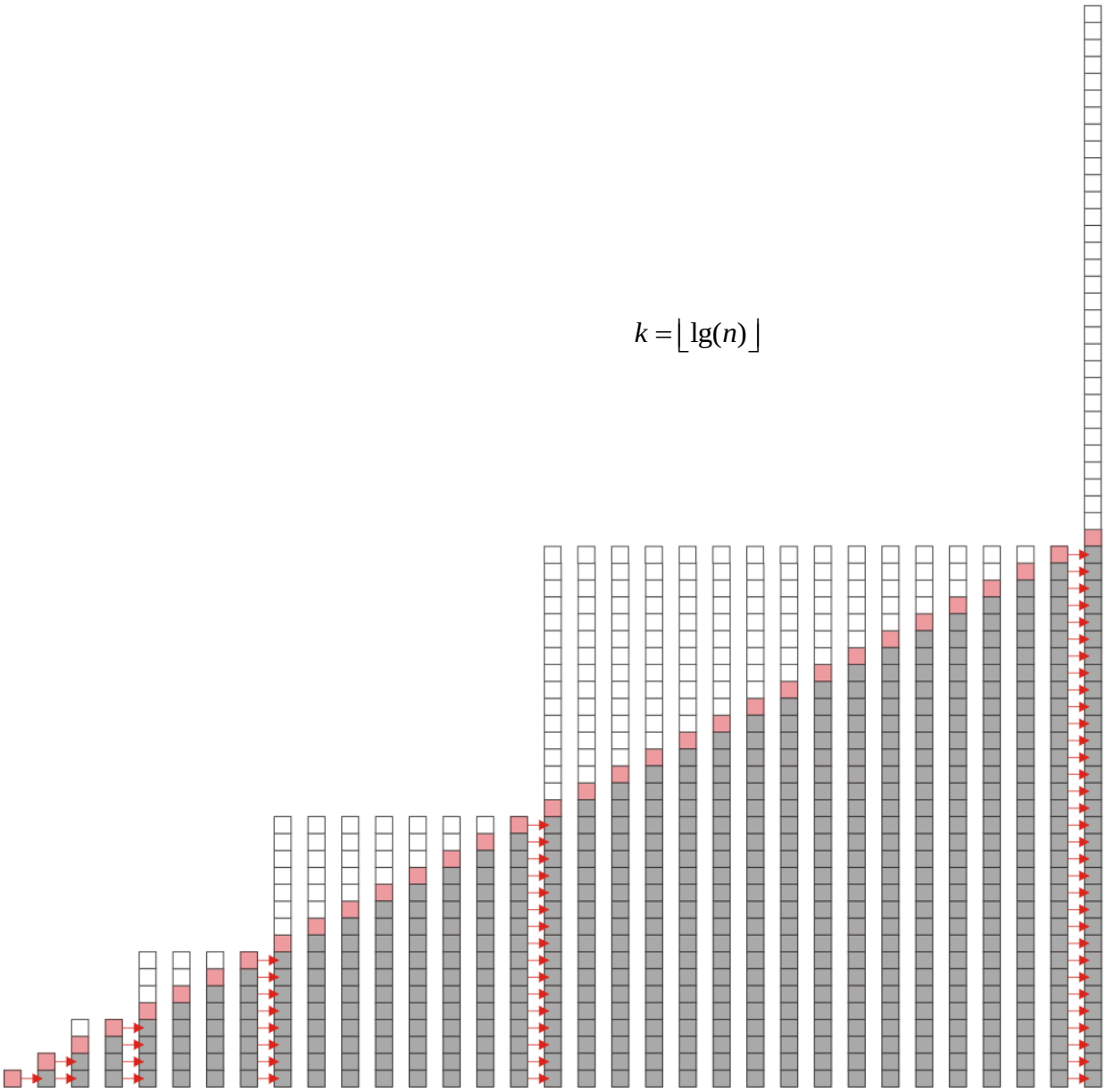

Figure 3. The number of copies required to insert *n* objects when doubling the capacity.

In order to keep things simple, let us assume we are inserting *n* objects. Thus, we will require 1, 2, 4, 8, ..., all the way up to 2^k where $k = \lfloor \lg(n) \rfloor$. Thus, we determine that we require

$$\begin{aligned}\sum_{k=0}^{\lfloor \lg(n) \rfloor} 2^k &= 2^{\lfloor \lg(n) \rfloor + 1} - 1 \\ &\leq 2^{\lg(n)+1} - 1 = 2^{\lg(n)} 2^1 - 1 = 2n - 1 = \Theta(n)\end{aligned}$$

copies. Because $n = 2^m$, this equals $n - 1$ copies. Thus, the amortized run time will be $\Theta\left(\frac{n-1}{n}\right) = \Theta(1)$. There will, however, be unused memory, and in the worst case, the unused memory at any time will be $n - 2 = \Theta(n)$ where n is the current number of objects in the array.

3.2.4.3 Discussion

It may appear to be absurd to increase the capacity of the array by 1—would it not be better to increase it, say, by 10 or 32? Unfortunately, while reduce the number of times that the capacity of the array has to be increased, the average number of copies per insertion will be, for these two examples, $n/10$ and $n/32$, respectively, and therefore, the amortized run time will still be $\Theta(n)$ per insertion.

3.2.5 Applications

We will look at three applications: parsing XML, parsing C++, and evaluating mathematical expressions written using the reverse Polish notation.

3.2.5.1 Parsing XML

XML (for example, XHTML) is a *markup language* that comprised of matching opening and closing tags, for example, `<p>` and `</p>`; `<html>` and `</html>`; and `<msup>` and `</msup>`. These tags must be nested so that they satisfy the rule that when a closing tag appears, it must match the most recent opening. An example of valid xhtml is

```
<html>
     <head><title>Hello</title></head>
     <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

It would not be valid to have, for example, `<i>How <u>are</i> you</u>?` to code *How <u>are</u> you*? The proper way to code this would be `<i>How <u>are</u></i><u> you</u>?`. The justification for this is that now a stack may be used to parse XML. When passing through (parsing) the code, each opening tag is pushed onto a stack and when a closing tag is encountered, it must be a matching tag for the opening tag currently on the top of the stack. If it matches, the top of the stack is popped. If it does not, the XML is invalid.

Other possible errors include:

1. Encountering a closing tag before any opening tag has been pushed onto the stack, or
2. Reaching the end of the file but still having a non-empty stack.

3.2.5.2 Parsing C++

Similar to parsing XML, delimiters must be paired in C++. This includes parentheses `()`, brackets `[]`, and braces `{}`. Like XML, delimiters must be nested—under no circumstances will you ever have `[...(...]...)`. This was a deliberate design decision and consequently, a stack can be used to ensure that delimiters are correctly matched.

Note: this does not happen in usual mathematics, for example, both (3, 7] and]3, 7] are notations used by mathematicians to denote a half-open interval all points $3 < x \leq 7$.

3.2.5.3 Function Calls

When a function returns, it always goes back to the previous function from which it was called. For example, if `f` is a function that calls `g` and `g` then calls `h`, it will never happen that `h` returns to `f`. It must always return to `g` and when `g` returns, it will return to `f`, continuing wherever the function call in `f` was made. This is a stack behaviour and memory is allocated for each new function call on a stack. That is why it is possible to call a function recursively even if it has local variables:

- The first time the function is called, memory is allocated on a stack for the parameters and the local variables.
- If the function is called from within the function, new additional memory will be allocated to the second function call for all of the parameters and local variables.

Therefore, evaluating the factorial function using

```
double factorial( int n ) {
    return ( n <= 1 ) ? 1.0 : n * factorial( n - 1);
}
```

requires not only $\Theta(n)$ time, but also $\Theta(n)$ memory. It would be more efficient to use:

```
double factorial( int n ) {
    double result = 1.0;

    for ( int i = 2; i < n; ++i ) {
        result *= i;
    }

    return result;
}
```

This still uses $\Theta(n)$ time, but it only requires $\Theta(1)$ time (8 bytes for the `double result` and 4 for the `int i`).

3.2.5.4 Reverse Polish Expressions

Reverse Polish is a means for writing mathematical expressions by writing the operands first and then the operator. For example, 3 4 + is normally what we write as 3 + 4. The expression 3 4 5 + × is interpreted by evaluating 4 5 + first, yielding 9, which simplifies the expression to 3 9 × which equals 27. This is different from 3 4 × 5 + where we evaluate 3 4 × first (which equals 12), and then 12 5 + evaluates to 17.

Notice that no parentheses were needed 3 × 4 + 5, by order of operations, always equals 17 and parentheses are required to have it equal 27: 3 × (4 + 5).

Reverse Polish is important for other reasons: this is how a CPU behaves. You must load both operands into registers before you can apply an operation. Later, in ECE 351 *Compilers*, computer engineering students will see how expressions such as

```
double result = (a + b)*c - d*e + f + (g - h)/i;
```

are parsed and converted into the corresponding assembly instructions. An example of a reverse-polish programming language is postscript:

```
0 10 360 {                  % Go from 0 to 360 degrees in 10-degree steps
    newpath                 % Start a new path
    gsave                   % Keep rotations temporary
        144 144 moveto
        rotate              % Rotate by degrees on stack from 'for'
        72 0 rlineto
        stroke
    grestore                % Get back the unrotated state
} for % Iterate over angles
```

creates the graphic shown in Figure 9.

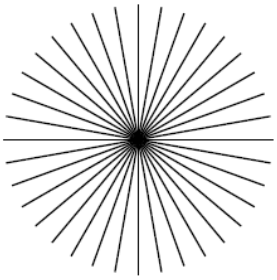

Figure 9. The output of the sample postscript code.

The postscript code is taken from http://www.tailrecursive.org/postscript/examples/rotate.html.

We will, however, see how a stack can be used to evaluate a reverse-Polish expression.

We will proceed as follows:

1. Start with an empty stack,
2. Read through the reverse-Polish expression from left to right and if you encounter
    a. an operand, push it onto the stack, and
    b. an operator •, pop the last objects off the stack, first *b* and then *a*, evaluate *a* • *b*and push the result back onto the stack.
3. Continue until you get to the end of the expression.

At the end, the stack should contain exactly one value: the solution. The is an error in the reverse-Polish expression if:

1. An operator is encountered when there is either one or zero objects on the stack, and
2. The expression is entirely parsed but there are still two or more objects on the stack.

Consider evaluating:

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

First, 1, 2 and 3 are placed on the stack (the top of the stack is to the right):

1	2	3				

On encountering +, we pop 3 and 2, evaluate 2 + 3 = 5 and push 5 onto the stack:

1	5					

Next, we push 4, 5, and 6 onto the stack:

1	5	4	5	6		

We encounter ×, pop 6 and 5, evaluate 5 × 6 = 30 and push that onto the stack:

1	5	4	30			

Next we encounter –, so we pop 30 and 4 and evaluate 4 – 30 = –26:

1	5	–26				

We push 7 onto the stack:

1	5	–26	7			

We encounter ×, so we pop 7 and –26 and calculate –26 × 7 = –182:

1	5	–182				

We encounter +, so we pop –182 and 5and calculate 5 + (–182) = –177:

1	–177					

We encounter –, so we pop –177 and 1 and calculate 1 – (–177) = 178:

178						

We push 8 and 9 onto the stack:

178	8	9				

Encounter a ×, pop twice and evaluate 8 × 9 = 72:

178	72					

Finally, we encounter +, pop twice, evaluate 178 + 72 = 250 and push that back onto the stack:

250						

Therefore, the expression evaluates to 250.

3.2.5.5 Robert's Rules of Order

This is only for those students interested in politics. Given a deliberative assembly, a group of people who are meeting to discuss and debate a particular topic, it is necessary to keep some form of order. Robert's Rules is a simplification of Parliamentary Rules used in the legislatures in Canada, the United Kingdom, and the United States. It is also quite reasonable as a guide for smaller organizations. This book, while written in the 1876, essentially uses a stack to keep track of the debate. There is a list of motions that can be made, and the motions are given precedence. For the assembly to begin discussing a motion, the motion must be pushed onto the top of an empty stack. The rules are very simple:

1. You are only allowed to discuss whatever is currently on top of the stack, and
2. For almost all motions, you can only pop a motion off the stack is to vote on it.

For example, someone in EngSoc could make the *main* motion that "EngSoc should spend $2,500 on a new computer." That motion would be placed on the stack. Anyone who wishes to speak out, either for or against it, may do so, regulated by a meeting chair. One motion of higher precedence is amending the main motion. Another student might make the motion "I motion that we amend the main motion to read that 'EngSoc should spend $1,500 on a new computer.'" At this point, the amendment is placed onto the stack. The **only** issue that may be discussed is the validity of the amendment. If a person was to get up and speak against the idea of even buying a computer in the first place, the chair of the meeting should rule that person *out of order*; that is, they are discussing a motion that is not currently at the top of the stack. The only *pop* the motion to amend is to vote on it. If the motion to amend passes, the main motion is amended and the discussion continues on the amended main motion. If the motion to amend fails, the discussion returns to a discussion over the original main motion.

By using a stack, Robert's Rules of Order allows everyone to understand what is currently being discussed and it keeps members from straying too far from the point at hand.

3.2.6 Stacks in the STL

The stack class in the STL has the following definition:

```
template <typename T>
class stack {
    public:
        stack();                  // not quite true...
        bool empty() const;
        int size() const;
        const T & top() const;
        void push( const T & );
        void pop();
};
```

An example of this stack in use is given here:

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> istack;
    istack.push( 13 );
    istack.push( 42 );

    std::cout << "Top: " << istack.top() << std::endl;
    istack.pop();                             // no return value

    std::cout << "Top: "  << istack.top()  << std::endl;
    std::cout << "Size: " << istack.size() << std::endl;

    return 0;
}
```