IV. Stacks

A. Introduction

1. Consider the 4 problems on pp. 170-1:

   (1) Model the discard pile in a card game

   (2) Model a railroad switching yard

   (3) Parentheses checker

   (4) Calculate and display base-two representation

Remainders are generated in right-to-left order. We need to "stack" them up, then print them out from top to bottom.


In summary ... a ____________________________ structure.

2. Definition of a stack as an ADT (abstract data type):

   A stack is: **an __________________ collection of data items in which_____________________________________________________.**

   Its basic operations are:

   1. ____________________________

   2. **Check if stack ____________________________

   3. _____: _____ an element __________________ of the stack

   4. _____: __________ the ______________ of the stack

   5. _____: __________ the ______________ of the stack

The terminology comes from a spring-loaded stack of plates in a cafeteria:
- Adding a plate pushed those below it are pushed down in the stack
- When a plate is removed from the stack, those below it pop up one position.
3. If we had a stack class we could use it to easily develop a short program for the base-conversion problem.

(See pp. 171-2 for the algorithm.)

/* Program that uses a stack to convert the base-ten
   * representation of a positive integer to base two.
   *
   * Input: A positive integer
   * Output: Base-two representation of the number
   *********************************************************

#include "Stack.h" // our own -- <stack> for STL version
#include <iostream>
using namespace std;

int main()
{
    unsigned number,       // the number to be converted
        remainder;    // remainder when number is divided by 2

    char response;         // user response

    do
    {
        cout << "Enter positive integer to convert: ";
        cin >> number;

        while (number != 0)
        {
            remainder = number % 2;

            number /= 2;
        }

        cout << "Base two representation: ";

        while (________________________________________);
        {

        }

        cout << remainder;
    }

    cout << endl;
    cout << endl;
    cout << "\nMore (Y or N)? ";
    cin >> response;
} while (response == 'Y' || response == 'y');
B. Building a Stack Class

Two steps:
1. __________ the class; and
2. __________ the class.

1. Designing a Stack Class

Designing a class consists of identifying those operations that are needed to manipulate the "real-world" object being modeled by the class. Time invested in this design phase pays off, because it results in a well-planned class that is easy to use.

Note: The operations are described __________________________. At this point, we have no idea what data members will be available, so the operations must be described in some way that is does not depend on any particular representation of the object.

The resulting specification then constitutes the "blueprint" for building the class.

From definition of stack as ADT, we must have (at least) the following operations:
- __________: Initializes an empty stack.)
- _______ operation: Examines a stack and return false or true depending on whether the stack contains any values:
- _______ operation: Modifies a stack by adding a value at the top of the stack:
- _______ operation: Retrieves the value at the top of the stack:
- _______ operation: Modifies a stack by removing the value at the top of the stack:

To help with debugging, add early on:
- _______ : Displays all the elements stored in the stack.

2. Implementing a Stack Class

Two steps:
1. Define __________________________
2. Define the ________________________

a. Selecting Data Members.

A stack must store a collection of values, so we begin by considering what kind of storage structure(s) to use.

Possibility #1:
Use an array with the top of the stack at position 0.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>89</td>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+ features: __________________________
- features: __________________________
Attempt #2 — A Better Approach

Instead of modeling the stack of plates, model a stack of ______________.

Keep the bottom of stack at position 0 and maintain a "pointer" myTop to the top of the stack.

e.g., Push 75, Push 89, Push 64, Pop

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

myTop = ____ myTop = ____ myTop = ____ myTop = ____ myTop = ____

Note: _______________________________________

So, we can begin the declaration of our class by selecting data members:

- Provide an ______________ data member to hold the stack elements.
- Provide a ______________ data member to refer to the ______________.
- Provide an ______________ data member to indicate the ______________.

Problems: We need an array declaration of the form

```c
ArrayElementType myArray[ARRAYCAPACITY];
```

- What type should be used?
  
  Solution (for now): Use the ______________ mechanism:
  
    ```c
    // put this before the class declaration
    ```

- What about the capacity?
  
  ```c
  // put this before the class declaration
  ```

- Then declare the array as a data member in the private section:
  
  ```c
  ```

Notes:
1. The typedef makes StackElement a ______________ for int. Putting it outside the class makes it accessible throughout the class and in any file that #includes Stack.h. If in the future we want a stack of reals, or characters, or ..., we need only change the typedef:

   ```c
typedef double StackElementType;
```
   or
   ```c
typedef char StackElementType;
```
   or...

   When the class library is recompiled, the type of the array's elements will be double or char or...
2. A more modern alternative that doesn't require using and changing a `typedef` is to use the ______________ mechanism to build a `Stack` class whose element type is left unspecified. The element type is then ______________ at compile time. We'll describe this soon. This is the approach used in the ______________.

3. Putting the `typedef` and declaration of `STACK_CAPACITY` ahead of the class declaration makes these declarations easy to find when they need changing.

4. If the type `StackElement` or the constant `STACK_CAPACITY` were defined as public members inside the class declaration, they could be accessed outside the class but would require qualification:

   ______________

   ______________

5. If we were to make the constant `STACK_CAPACITY` a class member we would probably make it a __________ data member:

   ______________ const int `STACK_CAPACITY` = 128;

   This makes it a property of the class useable by all class objects, but they do ______________ ______________ of `STACK_CAPACITY`.

So, we can begin writing `Stack.h`:

```
Stack.h

/* Stack.h provides a Stack class. */
* 
* Basic operations: 
* Constructor: Constructs an empty stack 
* empty: Checks if a stack is empty 
* push: Modifies a stack by adding a value at the top 
* top: Accesses the top stack value; leaves stack unchanged 
* pop: Modifies a stack by removing the value at the top 
* display: Displays all the stack elements 
* Class Invariant: 
* 1. The stack elements (if any) are stored in positions 
* 2. -1 <= myTop < STACK_CAPACITY

#ifndef STACK
#define STACK


#endif
```

class Stack
{

/***** Function Members *****/
public:

private:

}; // end of class declaration

b. Function Members

• Constructor:
  Simple enough to inline? ____________
  
  class Stack
  {
  public:
  /* --- Constructor ---
  Precondition: A stack has been declared.
  Postcondition: The stack has been constructed as an
  empty stack.
  -----------------------------*/
  
  
  }; // end of class declaration

A declaration
  Stack S;
will construct S as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>...</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• empty:
  Receives Stack containing it as a function member (implicitly)
  Returns: True if stack is empty, false otherwise.
  Member function? ____________
  const function? (Shouldn’t alter data members?) ____________
  Simple enough to inline? ____________
class Stack
{
    public:
        ... 
    /* --- Is the Stack empty? --- 
    * Receive: stack containing this function (implicitly) 
    * Returns: true if the Stack containing this function is empty 
    *           and false otherwise 
    *****************************************************************/

}; // end of class declaration

Test driver:

#include <iostream>
using namespace std;
#include "Stack.h"

int main()
{
    Stack s;
    cout << boolalpha << "s empty? " << s.empty() << endl;
}

• push:

    Receives: Stack containing it as a function member (implicitly) 
    Value to be added to stack 
    Returns: Modified Stack (implicitly) 

    Member function? ________
    const function? __________
    Simple enough to inline? ________

class Stack
{
    public:
        ... 
    /* --- Add a value to the stack --- 
    * 
    * Receive: The Stack containing this function (implicitly) 
    * A value to be added to a Stack 
    * Pass back: The Stack (implicitly), with value added at its 
    * top, provided there's space 
    * Output: "Stack full" message if no space for value 
    *****************************************************************/

}; // end of class declaration
Definition (in Stack.cc):

```cpp
void Stack::push ____________________________
{
    // or simply, ______________ = value;
    else
        cerr << "*** Stack is full -- can't add new value ***\n";
        << "Must increase value of STACK_CAPACITY in Stack.h\n";
}
```

Add at bottom of driver:

```cpp
for (int i = 1; i <= 128; i++) s.push(i);
cout << "Stack should now be full\n";
s.push(129);
```

**Output**

s empty? 1
Stack should now be full
*** Stack is full -- can't add new value ***

- **Output:**
  So we can test our operations.
  Receives: Stack containing it as a function member (implicitly)
  Output: Contents of Stack, from the top down.
  Member function? Yes
  const function? (Shouldn't alter data members?) Yes
  Simple enough to inline? No

**Prototype:**

```cpp
/* --- Display values stored in the stack ---
*
* Receive: The Stack containing this function (implicitly)
* The ostream out
* Output: The Stack's contents, from top down, to out
*******************************************************************************/

void display(ostream & out) const;
```

**Definition in Stack.cpp:**

```cpp
void Stack::display(ostream & out) const
{
}
```
Modify driver:
/*
for (int i = 1; i <= 128; i++) s.push(i);
   cout << "Stack should now be full
n";
s.push(129);
*/
for (int i = 1; i <= 4; i++) s.push(2*i);
cout << "Stack contents:\n";
s.display(cout);
cout << "boolalpha << s empty? " << s.empty() << endl;

Output
s empty? true
Stack contents:

* top:
   Member function? ______
   const function? ______
   Simple enough to inline? Probably not

Prototype:
/* --- Return value at top of the stack ---
 * * Receive: The Stack containing this function (implicitly)
 * * Return: The value at the top of the Stack, if nonempty
 * * Output: "Stack empty" message if stack is empty
 ***********************************************************/

Definition (in Stack.cpp):
StackElement Stack::top() const
{

}

Add to driver at bottom:
cout << "Top value: " << s.top() << endl;

Output
Stack contents:
8
6
4
2
s empty? false
Top value: 8
• pop:

Member function? 
const function? 
Simple enough to inline?

Prototype:
/* --- Remove value at top of the stack ---
 * 
 * Receive: The Stack containing this function (implicitly)
 * Pass back: The Stack containing this function (implicitly)
 *           with its top value (if any) removed
 * Output:  "Stack-empty" message if stack is empty.
 **********************************************************/

Definition (in Stack.cpp):

```cpp
void Stack::pop()
{
  ___________________________  // Preserve stack invariant
  ___________________________
  else
    cerr << "*** Stack is empty -- can't remove a value ***\n";
}
```

Add to driver at bottom:
while (!s.empty())
{
  cout << "Popping " << s.top() << endl;
  s.pop();
}  
cout << "s empty? " << s.empty() << endl;

Output
Stack contents:
8
6
4
2
s empty? false
Top value: false
Popping 8
Popping 6
Popping 4
Popping 2
s empty? true
C. Two Applications of Stacks

Use of Stacks in Function Calls

Whenever a function is begins execution (i.e., is activated), an_________________________ (or stack frame) is created to store the current environment for that function. Its contents include:

- parameters
- caller’s state information (saved)
  - (e.g., contents of registers, return address)
- local variables
- temporary storage

What kind of data structure should be used to store these when a function calls other functions and interrupts its own execution so that they can be recovered and the system reset when the function resumes execution?

Clearly need ______________ behavior. (Obviously necessary for recursive functions.)

So use a _________. Since it is manipulated at run-time, it is called the _______________________________.

What happens when a function is called:

1. _______________________________________________________________

2. Copy its arguments into the parameter spaces

3. Transfer control to the address of the function’s body

So the________________________ in the run-time stack is always that of the function ______________________.

What happens when a function terminates?

1. _______________________________________________________________

2. Use new top activation record to ___________________________ from the run-time stack

   execution of it.

Examples:

```c
int main()
{
    f2(...);
    f3(...);
}

void f1(...) { . . .
void f2(...) { . . . f1(...); . . .
void f3(...) { . . . f2(...); . . .
```
int factorial(int n)
{ if (n < 2)
    return 1;
else
    return n * factorial(n-1);
}

What happens to the run-time stack when the following statement executes?
int answer = factorial(4);

This pushing and popping of the run-time stack is the real ______________ associated with function calls that ______________ functions avoids by replacing the function call with the body of the function.

**Application to Reverse Polish Notation**

1. What is RPN?
   A notation for arithmetic expressions in which ___________________________________________________________.
   Expressions can be written _____________________________________________.
   Developed by Polish logician, Jan Lukasiewics, in 1950's

   _______ notation: operators written _________ the operands
   _______ " ( _____ )": operators written _________ the operands
   _______ " : operators written _________ the operands

   Examples:

<table>
<thead>
<tr>
<th>INFIX</th>
<th>RPN (POSTFIX)</th>
<th>PREFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>A B +</td>
<td>+ A B</td>
</tr>
<tr>
<td>A * B + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A * (B + C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A - (B - (C - D))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A - B - C - D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Evaluating RPN Expressions
   a. "By hand": Underlining technique:
      
      Scan the expression from left to right to find an operator. Locate ("underline") the last two preceding operands and combine them using this operator. Repeat this until the end of the expression is reached.
      
      Example: \[ 2 \ 3 \ 4 \ + \ 5 \ 6 \ - \ - \ * \]
      
      \[ \rightarrow \ 2 \ 3 \ 4 \ + \ 5 \ 6 \ - \ - \ * \ \rightarrow \ 2 \ \underline{5} \ 6 \ - \ - \ * \]
      
      \[ \rightarrow \ 2 \ 7 \ 5 \ 6 \ - \ - \ * \ \rightarrow \ 2 \ 7 \ \underline{-} \ - \ * \]
      
      \[ \rightarrow \ 2 \ 7 \ -1 \ - \ * \ \rightarrow \ 2 \ \underline{-} \ * \ \rightarrow \ 2 \ 8 \ * \rightarrow \ \]

   b. Algorithm — using a stack of operands (p. 195)
      
      Receive: An RPN expression.
      Return: The value of the RPN expression (unless an error occurred).
      Note: Uses a stack to store operands.

      1. Initialize an empty stack.
      2. Repeat the following until the end of the expression is encountered:
         a. Get the next token (constant, variable, arithmetic operator) in the RPN expression.
         b. If the token is an operand, push it onto the stack. If it is an operator, then do the following:
            (i) Pop the top two values from the stack. (If the stack does not contain two items, an error due to a malformed RPN expression has occurred, and evaluation is terminated.)
            (ii) Apply the operator to these two values.
            (iii) Push the resulting value back onto the stack.
      3. When the end of the expression is encountered, its value is on top of the stack (and, in fact, must be the only value in the stack).

      Example: See p. 196.

      To generate code, change (ii) and (iii) to:
      (ii') Generate code: \[ \text{LOAD } \text{operand}_1 \]
      \[ \text{op } \text{operand}_2 \]
      \[ \text{STORE } \text{TEMP} \# \]

      Example: Generate code for \[ A \ B \ + \ C \ D \ + \ * \]

   c. Unary minus causes problems:
      
      Example: \[ 5 \ 3 \ - \ - \rightarrow \]
      
      \[ 5 \ 3 \ - \ - \rightarrow \]
      
      We'll use a different symbol:
3. Converting from Infix to RPN
   a. "By hand": Represent infix expression as an expression tree:
      \[
      \begin{align*}
      A & \ast B + C \\
      A \ast (B + C) & \\
      \frac{(A + B) \ast C}{D - E}
      \end{align*}
      \]
      Traverse the tree in **Left-Right-Parent** order to get ____________
      Traverse tree in **Parent-Left-Right** order to get ____________
      Traverse tree in **Left-Parent-Right** order to get ____________ [must insert (')s]
   
   b. By hand: "Fully parenthesize-move-erase" method:
      1. Fully parenthesize the expression.
      2. Replace each right parenthesis by the corresponding operator.
      3. Erase all left parentheses.
      
      Examples:
      \[
      \begin{align*}
      A \ast B + C & \rightarrow (A \ast B) + C \\
      (A \ast B) + C & \rightarrow (A B \ast C) + \\
      A B \ast C + & \\
      A \ast (B + C) & \rightarrow \\
      \end{align*}
      \]
      \[
      \begin{align*}
      (A + B) \ast C) / (D - E) & \rightarrow 
      \end{align*}
      \]
   
   c. Algorithm — using a stack of operators (See pp.199-201)