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.

Need a "last-discarded-first-removed," "last-pushed-onto-first-removed," "last-stored-first-removed, " "last-generated-first-displayed" structured data type.

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 ____

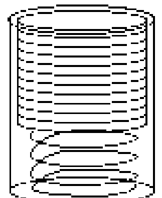
3. ________: _______ an element _______________________ of the stack

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 << "\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 payd 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 ________________________

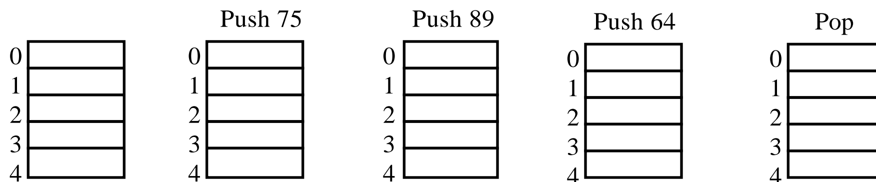
a. Selecting Data Members.

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.

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



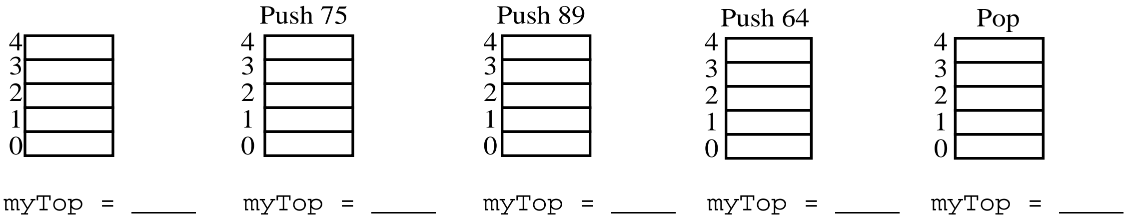
+ features: ____

– 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



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

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

— What type should be used?

Solution (for now): Use the ____________________ mechanism:

```
______________________________________________
// put this before the class declaration
```

— What about the capacity?

```
______________________________________________
// put this before the class declaration
```

— Then declare the array as a data member in the private section:

____

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`:

```
typedef double StackElementType;
```
or
```
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
 *      0, 1, . . ., myTop of myArray.
 *   2. -1 <= myTop < STACK_CAPACITY
 ------------------------------------------------------------------*/

#ifndef STACK
#define STACK

_______________________________________

_______________________________________
```

```
class Stack
{

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

/***** Data Members *****/
private:

  ____________________________________

  ____________________________________

}; // end of class declaration
 . . .
 #endif
```

b. Function Members

- *Constructor* :

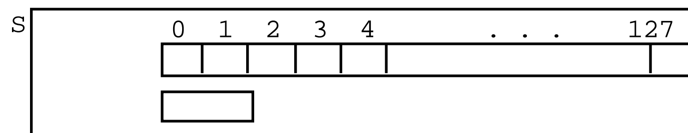
  Simple enough to inline? ____________

```
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:



- *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

_________________________________

_________________________________
```

<u>Test driver:</u> <u>Output</u>

```
#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`):

```
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:

```
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:

```
/* --- 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:

```
void Stack::display(ostream & out) const
{






}
```

<u>Modify driver</u>:

```
/*
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;
```

<u>Output</u>

```
s empty? true
Stack contents:
```

- *top:*

Member function? _____

`const` function? _____

Simple enough to inline? <u>Probably not</u>

<u>Prototype</u>:

```
/* --- 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
 ************************************************************/
```

____

<u>Definition (in `Stack.cpp`):</u>

```
StackElement Stack::top() const
{




}
```

<u>Add to driver at bottom</u>:

```
cout << "Top value: " << s.top() << endl;
```

<u>Output</u>

```
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`):

```
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: 8
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) ____from the run-time stack

(2) Use new top activiation record to ____ execution of it.

Examples:

```
. . .
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:

INFIX	RPN (POSTFIX)	PREFIX
A + B	A B +	+ A B
A * B + C		
A * (B + C)		
A - (B - (C - D))		
A - B - C - D		

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 - - *`

```
2 3 4 + 5 6 - - *      2 __ 5 6 - - *

2 7 5 6 - - *          2 7 ____ - *

2 7 -1 - *        2 __ *         2 8 *      [ ]
```

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:
```
LOAD   operand1
 op    operand2
STORE  TEMP#
```
(iii') Push `TEMP#` onto stack.

Example: Generate code for A B + C D + *

c. Unary minus causes problems:

Example: `5 3 - -`

`5 3 - -`

We'll use a different symbol:

3. Converting from Infix to RPN

a. "By hand": Represent infix expression as an *expression tree***:**

`A * B + C`   `A * (B + C)`   `((A + B) * C) / (D - E)`

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:

`A * B + C`   `((A * B) + C)`   `((A B * C +`   `A B * C +`

`A * (B + C)`

`((A + B) * C) / (D - E)`

c. Algorithm — using a stack of operators (See pp.199-201)