Stacks

Linear data structures

- Collection of components that can be arranged as a straight line
- Data structure grows or shrinks as we add or remove objects
- ADTs provide an abstract layer for various operations on these data structures
  - ADTs contain a collection of data structures together with a set of operations defined on those data structures
  - Data structures are usually composite data structures

The logical level

- What is a stack?
  - A data structure in which elements are added and removed from only one end
    - The remainder of the stack remains undisturbed
  - A "last in first out" (LIFO) structure
  - Used for processing nested structures, or manage algorithms that call subprocesses
    - Think of how a function gets executed
      - Stack of *activation record* of functions
    - Also think of top-down design technique
- Operations on stack
  - Stack is a dynamic data structure and changes as the elements are added to or removed from it
  - Stack ADT specification

    Structure. Elements are added to and removed from the top of the stack
    Create stack. Initializes a stack to an empty state

    ```
    stack_type create_stack ( stack_type stack );
    ```

    Function. Initializes stack to an empty state
    Input. None
    Preconditions. None
    Output. A stack
    Postconditions. Stack is empty

    Destroy stack. Remove the stack's existence
    Function. Removes all elements from stack, leaving the stack empty
    Input. A stack
    Preconditions. The stack exists
    Output. Stack
    Postconditions. The stack is empty

    Empty stack? Checks to see if the stack is empty
    Function. Tests whether the stack is empty
    Input. A stack
    Preconditions. Stack has been created
    Output. True or False (1 or 0)
    Postconditions. True if stack is empty, false otherwise

    Full stack? Checks to see if the stack is full
    Function. Tests whether the stack is full
    Input. A stack

Preconditions. Stack has been created
Output. True or False (1 or 0)
Postconditions. True if stack is full, false otherwise

Push an element. Push an element into the stack

Function. Adds a news element to the top of the stack
Input. Stack as well as the new element
Preconditions. Stack has been created and is not full
Output. Stack
Postconditions. The new element is added to the top of the stack

Pop an element. Pop an element from the stack

Function. Removes the top element from the stack and returns it in the popped element
Input. Stack
Preconditions. Stack has been created and is not empty
Output. Stack, popped element
Postconditions. Stack is returned with the topmost element removed which is put into the popped element

The user level

- What happens if we try to read a float when the user inputs a nonnumeric character? Should we continue, or should we print an error message?
- Write a function `read_float`

Function. `float read_float ( char * data, int * error )`
Read characters representing a floating point number from string `data`, convert them to a floating point number; the character stream is terminated by a blank; any other nonnumeric character (except decimal point) generates an error

Input. String `data`

Preconditions. `data` contains the floating point number; newline is considered as a blank character

Output. Floating point number is returned; error is flagged as non-zero

Postconditions. Error is true if number terminates with a nonblank; If error occurs, the number is returned as zero; otherwise, the number is the floating point representation of the string of characters

- Algorithm for `read_float`
  - Scan the string one character at a time until a non-numeric character is encountered
  - Convert the "whole" part of the number
  - If decimal point is found, convert the "decimal" part of the number
  - Add the whole and decimal parts
- Convert whole part
  - Algorithm

    number ← 0.0
    read a character from `data`
    while character is numeric
      number ← (10 × number) + numeric equivalent of character
      read next character from `data`

  - When the loop terminates, number contains the whole part while character contains the non-numeric character, leaving three possibilities
    1. character is blank ⇒ no decimal part
       return the floating point number and set error to zero
    2. character is a decimal point ⇒ read and convert the decimal part of the number; no error so far

3. character is any other character ⇒ set error to true

– Conditions 1 and 3 terminate the function while condition 2 requires more work
– Need algorithm to read and convert the decimal part

- Convert fractional part
  - Algorithm (first pass)

```
frac ← 0.0
read a character from data
while character is numeric
   frac ← ( frac ÷ 10 ) + numeric equivalent of character
   read next character from data
frac ← frac ÷ 10
```

  - Now try the algorithm with a sequence of digits 567 (should result in .567 but gives .765)
  - Problem – We should have reversed the sequence of characters
  - Solution – Use a stack or LIFO
- Convert fractional part – Read and push
  - Algorithm

```
create_stack
read a character from data
while character is numeric
   push character into stack
   read next character from data
```

  - This has the effect of putting all the numeric characters into stack
- Convert fractional part – Pop and calculate
  - Algorithm

```
while stack is not empty
   pop a character
   frac ← ( frac ÷ 10 ) + numeric equivalent of character
frac ← frac ÷ 10
```

- Add whole and decimal part

```
float_num ← number + frac
```

- What if the number is negative?
  - Multiply the result by -1 to get the correct sign
- Additional error checking
  - Set error to 1 if the decimal part terminates with a non-numeric character
  - Destroy the stack in this case
- Final function

```
/*************************************************************************/
/* read_float : Read a floating point number from a character string, by  */
/*   reading characters until a blank or newline character is encountered, */
/*   and convert the characters into equivalent digits.  Error is set if the */
/*   string is terminated by a non-numeric character (other than blank)   */
```

```
/*****************************************************************************/

#define ZERO '0'      /* Character zero                                      */
#define MINUS_SIGN '-'
#define NEWLINE    '\n'
#define DECIMAL_PT '.'

float read_float ( char * data, int * error )
{
    float number,     /* whole part of the floating point number       */
          frac;       /* fractional part of the floating point number  */
    int   sign;       /* 1 if positive, -1 if negative                 */
    char  digit;      /* character from of digit                       */
    stack_type stack; /* stack of characters                           */
    int   i;          /* Index in the character string                 */

    /* Nested function ctod -- Character to digit                        */

    int ctod ( char digit )
    {
        return ( digit - ZERO );
    }

    /* Initialization phase                                              */

    *error = 0;
    number = frac = 0.0;

    /* Skip leading spaces (including tabs)                              */

    for ( i = 0; ( i < strlen ( data ) && isspace ( data[i] ) ); i++ );

    /* Determine if number is positive or negative                       */

    sign = ( data[i] == MINUS_SIGN ) ? -1 : 1;
    if ( sign == -1 )
        i++;

    /* Calculate whole part of the floating point number                 */

    while ( ( i < strlen ( data ) ) && isdigit ( data[i] ) )
        number = ( 10 * number ) + ctod ( data[i++] );

    /* Check for termination of the whole part with space                */
    /* Return if terminated after fixing the sign                        */

    if ( isspace ( data[i] ) )
        return ( sign * number );

    /* Check if the next character is a decimal point                    */

    if ( data[i] != DECIMAL_PT )
    {
        *error = 1;
```

```
        return ( sign * number );
    }

    /* There is a decimal point                                    */

    create_stack ( stack );      /* Create the stack               */
    i++;                         /* Move the index forward         */

    /* Put numbers into the stack                                  */

    while ( ( i < strlen ( data ) ) && isdigit ( data[i] ) )
        push ( stack, data[i++] );

    /* Check for proper termination of fractional part             */

    if ( ! isspace ( data[i] )
    {
        *error = 1;
        destroy_stack ( stack );
        return ( sign * number );
    }

    /* Compute fractional part                                     */

    while ( ! empty_stack ( stack ) )
        frac = ( frac / 10 ) + pop ( stack );
    frac /= 10;

    return ( sign * ( number + frac ) );
}
```

- User level assumes the existence of stack manipulation function that are hidden in the stack ADT

The implementation level

- Implementation of a stack as a static array
  - All elements of a stack are the same type, allowing the stack to be implemented as an array
  - First element goes in position 0, second element in position 1, and so on
  - We need to keep track of the top of stack
  - Can be accomplished through a structure

```
#define MAX_STACK 100

typedef struct
{
    char elements[MAX_STACK];    /* Character type elements in the stack  */
    int  top;                    /* Top of stack                          */
} stack_type;
```

  - `top` should be initialized to 0, so that it always points to the first free element in the stack
- Stack operations with the array implementation
  - Creating the stack

```
stack_type * create_stack ( stack_type * stack )
{
    /* Allocate memory and initialize the top of stack        */

    stack = ( stack_type * ) malloc ( sizeof ( stack_type ) );
    stack -> top = 0;

    return ( stack );
}
```

– Destroying the stack

```
void destroy_stack ( stack_type * stack )
{
    free ( stack );
}
```

– Function to check for empty stack

```
int empty_stack ( stack_type * stack )
{
    return ( stack -> top == 0 );
}
```

– Function to check for stack being full

  * Being a dynamic data structure, stack cannot be full
  * We are limited to the maximum size of the array for stack due to our choice of implementation

```
int full_stack ( stack_type * stack )
{
    return ( stack -> top == MAX_STACK );
}
```

– Pushing an element into the stack

```
stack_type * push ( stack_type * stack, char element )
{
    stack -> elements[stack->top] = element;
    ( stack -> top )++;

    return ( stack );
}
```

  * We should check for the condition that the stack is not full
  * *Stack overflow* is the condition resulting from trying to push an element onto a full stack

```
#define OVERFLOW 1
stack_type * push ( stack_type * stack, char element, int * error )
{
    if ( full_stack ( stack ) )
    {
        *error = OVERFLOW;
        return ( stack );    /* Return stack without modification */
    }

    stack -> elements[stack->top] = element;
    ( stack -> top )++;

    return ( stack );
}
```

- Popping an element from the stack

```
char pop ( stack_type * stack )
{
    ( stack -> top )--;
    return ( stack -> elements[stack->top] );
}
```

  * We should check for the condition that the stack is not empty
  * *Stack underflow* is the condition resulting from trying to pop an element from an empty stack

```
#define UNDERFLOW 2
char pop ( stack_type * stack, int * error )
{
    if ( empty_stack ( stack ) )
    {
        *error = UNDERFLOW;
        return ( );    /* Return nothing */
    }

    ( stack -> top )--;
    return ( stack -> elements[stack->top] );
}
```

- More general implementation
  - The stack operations can be adapted to other data types more easily if we declare the following:

```
typedef char stack_element_type;
```

  - This allows us to generalize the stack operations by just changing the `char` to another data type as long as the stack operations are coded with `stack_element_type`

- Implementation of a stack as a linked structure
  - True dynamic memory allocation-based algorithm
  - Designing the structure
    * Need to create the space for each element on the fly
    * The space created for the element should also have room to point to another element, which has followed just behind
    * We'll call each element in this stack a *node*
    * Each node contains an information field and a field for connection with the rest of the stack
    * The entire node can be declared as follows

```
typedef struct stack_node_type
{
    stack_element_type     element;  /* The information field               */
    struct stack_node_type *next;    /* Pointer to the next element in stack */
} stack_type;
```

  - Creating the stack

```
stack_type * create_stack ( stack_type * stack )
{
    stack = ( stack_type * ) NULL;

    return ( stack );
}
```

  - Function to check for empty stack

```
int empty_stack ( stack_type * stack )
{
    return ( stack == NULL );       /* Returns true if stack is empty */
}
```

– Pushing an element into the stack

```
stack_type * push ( stack_type * stack, stack_element_type element )
{
    stack_type * node;        /* A node in the stack                    */

    node = ( stack_type * ) malloc ( sizeof ( stack_type ) );

    node -> element = element;
    node -> next = stack;
    stack = node;

    return ( stack );
}
```

– Popping an element from the stack

```
stack_element_type pop ( stack_type * stack )
{
    stack_element_type element;
    stack_type         *tmp;

    /* Save the element and the pointer to the top of the stack       */

    element = stack -> element;
    tmp = stack;

    /* Set stack top to point to the next element in the stack        */

    stack = stack -> next;

    /* Free the space occupied by the top node of the stack           */

    free ( tmp );

    return ( element );
}
```

– Pop with a check for stack underflow

```
#define UNDERFLOW 2
stack_element_type pop ( stack_type * stack, int * error )
{
    stack_element_type element;
    stack_type        *tmp;

    if ( empty_stack ( stack ) )
    {
        *error = UNDERFLOW;
        return ( );    /* Return nothing */
    }
```

```
    /* Save the element and the pointer to the top of the stack       */

    element = stack -> element;
    tmp = stack;

    /* Set stack top to point to the next element in the stack        */

    stack = stack -> next;

    /* Free the space occupied by the top node of the stack           */

    free ( tmp );

    return ( element );
}
```

– Destroying the stack

```
void destroy_stack ( stack_type * stack )
{
    int error;
    stack_element_type element;

    while ( ! empty_stack ( stack ) )
        element = pop ( stack, error );
}
```

- Comparing stack implementations (the two ADTs)
  - Compare the number of executable statements in each operation
  - Compare the space required
    - Array implementation requires the declaration of a fixed amount of space plus one integer for index
    - Linked implementation requires the actual amount of space used at run time but the node is larger (due to the `next` field)
  - Relative efficiency

Function	Array-based implementation	Pointer-based implementation
`create_stack`	$O(1)$	$O(1)$
`destroy_stack`	$O(1)$	$O(N)$
`empty_stack`	$O(1)$	$O(1)$
`full_stack`	$O(1)$	–
`push`	$O(1)$	$O(1)$
`pop`	$O(1)$	$O(1)$