ECE36800 Data structures

Stacks
Chapter 4 (pp. 127-150)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Stack definition
- Primitive operators
- Stack implementations
  - Linked list
  - Array of fixed size
  - Array of variable size

- Reference pages: pp. 127-250

Stacks

- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted only at one end, called the top of the stack
- Can also be called LIFO (Last In First Out)
- A stack of papers, for example
  - The stack is either empty or not empty
  - Only the top item is visible
  - Items are ordered (from bottom to top)
  - Items may be added or removed only at top
- Other examples:
  - A call stack of frames for function calls
  - A spring-loaded plate dispenser for buffet

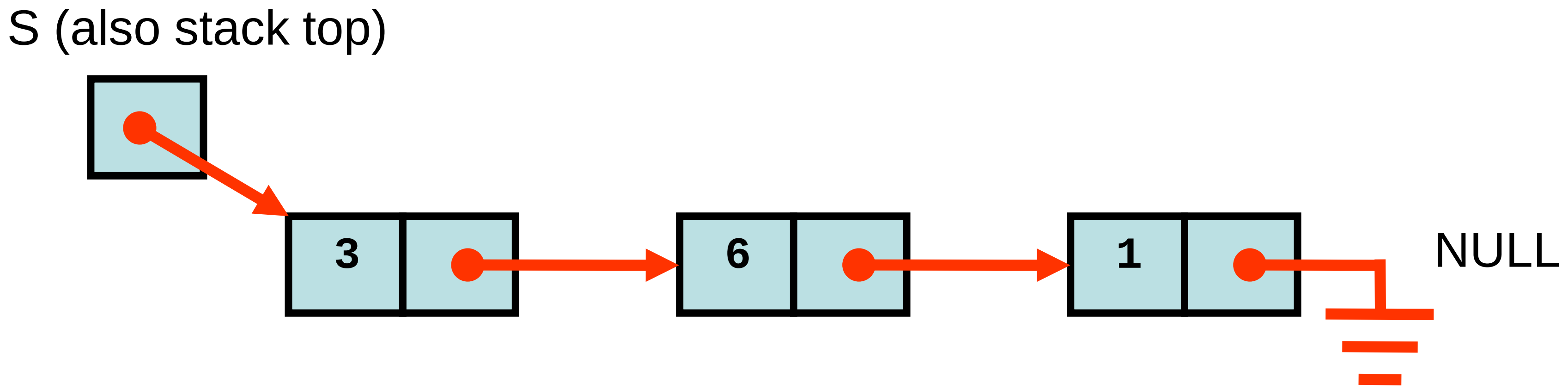
Primitive operators

- Push(S, i)
  - Add item i on the top of stack S
  - Must make sure it does not cause overflow
- Pop(S)
  - Get the top item of stack S, remove it from S and return it, e.g., i = Pop(S)
  - Must make sure that stack was not empty
- Stack_top(S)
  - Return the top item of stack S with no change to S, e.g., i = Stack_top(S)
  - Must make sure that stack was not empty
- Empty(S) (or Is_empty(S))
  - Return true if stack S is empty; false otherwise

Linked list implementation

- Seek O(1) time complexity for all primitive operations
- Empty list == Empty stack
- Head of list == Stack top
  - Meaningful only if list is not empty
- Insert at head = Push
  - To check for overflow, verify that the address returned by malloc is not NULL
- Remove at head = Pop
  - Meaningful only if list is not empty

Linked list example



3 = Pop(S)

Push(S, 4)

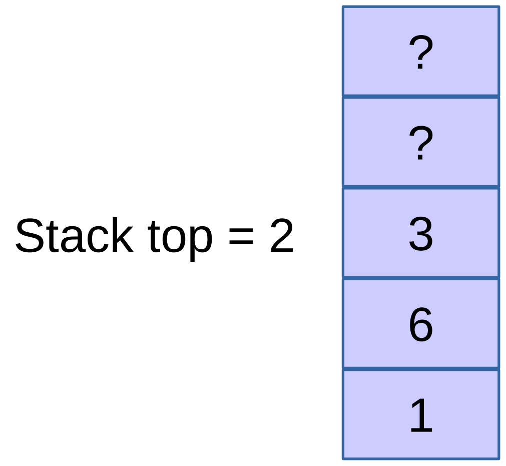
Array implementation

- Use a struct to store three fields:
  - Address of the array to store items
  - Total size of the array (the maximum number of items that can be on the stack)
  - Index of the stack top
- Assume that stack top index initialized to -1 for an empty stack
  - Empty stack == Stack top index -1
  - Stack top == item in array at location indexed by stack top (if not -1)
  - Push == Increment stack top index, store item in array at location indexed by stack top (if the stack top index < array size)
  - Pop == Store in a temporary variable the item in the array at location indexed by the stack top (if not -1), decrement stack top index, return value stored in the temporary variable

Array example (assume an array size of 5)

3 = Pop(S)

Push(S, 4)



C program for implementing a stack with an array of static size

- Slides from page 9 (this page) to page 23 not covered in class
- The size of the array for storing items on a stack is fixed in the program (called STACK_INT_SIZE), and not stored in the struct for stack
- Array is statically allocated within the struct

```
/*---------------------------------------
 * program: stack_int.h
 *         Implement a stack for integers
 * Programmer:  ece36800
 *--------------------------------------*/

#ifndef __stack_int__
#define __stack_int__

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define STACK_INT_SIZE 100
```

stack_int.h cont..

```
// declare stack structure
struct Stack_Int
{
  int Top;
  int Items[STACK_INT_SIZE];
};

typedef struct Stack_Int Stack_Int_t;


// function prototypes
void Stack_Int_Init(Stack_Int_t *S_Ptr);
int Stack_Int_Empty(Stack_Int_t *S_Ptr);
int Stack_Int_Stacktop(Stack_Int_t *S_Ptr);
int Stack_Int_Pop(Stack_Int_t *S_Ptr);
int Stack_Int_Push(Stack_Int_t *S_Ptr,
                     int Item);

#endif // __stack_int__
```

```c
/*----------------------------------------
 * program: stack_int.c
 * Programmer:  ece36800
 * Description: define stack functions
 *----------------------------------------*/

#include <stdio.h>
#include "stack_int.h"

void Stack_Int_Init(Stack_Int_t *S_Ptr)
{
  /* initialize the stack of integers */

  S_Ptr->Top = -1;

} /* Stack_Int_Init() */


int Stack_Int_Empty(Stack_Int_t *S_Ptr)
{
  /* is stack empty??  (TRUE/FALSE) */

  return (S_Ptr->Top < 0);

} /* Stack_Int_Empty() */
```

stack_int.c cont..

```
int Stack_Int_Push (
  Stack_Int_t *S_Ptr,
  int Item)
{
  /* place Item on top of stack */

  if (S_Ptr->Top >= STACK_INT_SIZE - 1)
  {
    return (FALSE);
  }
  S_Ptr->Items[++(S_Ptr->Top)] = Item;

  return (TRUE);

} /* Stack_Int_Push */
```

stack_int.c cont..

```
int Stack_Int_Stacktop(Stack_Int_t *S_Ptr)
{
  /* view and return item at top of stack */

  return (S_Ptr->Items[S_Ptr->Top]);

} /* Stack_Int_Stacktop() */


int Stack_Int_Pop(Stack_Int_t *S_Ptr)
{
  /* pull top item off stack */

  int Results = 0;

  Results = S_Ptr->Items[S_Ptr->Top];
  (S_Ptr->Top)--;

  return (Results);

} /* Stack_Int_Pop() */
```

```c
/*----------------------------------------
 * program: insert_at_bottom.c
 * Programmer:  ece36800
 * Description:
 *  develop function that would insert a new item at the bottom
 *   of the stack using only the public stack functions
 *----------------------------------------*/

#include <stdio.h>
#include "stack_int.h"

void Insert_At_Bottom (Stack_Int_t *S_Ptr, int Item)
{
  Stack_Int_t Aux_Stack; // for storing the contents of stack in reverse
  int Temp = 0;

  Stack_Int_Init(&Aux_Stack);
  while (!Stack_Int_Empty(S_Ptr)) // store contents in reverse
  {
    Temp = Stack_Int_Pop(S_Ptr);
    Stack_Int_Push(&Aux_Stack, Temp);
  }
  Stack_Int_Push(S_Ptr, Item); // new item at bottom
  while (!Stack_Int_Empty(&Aux_Stack)) // put back the rest
  {
    Temp = Stack_Int_Pop(&Aux_Stack);
    Stack_Int_Push(S_Ptr, Temp);
  }
} /* Insert_At_Bottom() */
```

insert_at_bottom.c cont..

```
int main(void)
{
  Stack_Int_t Stack;

  Stack_Int_Init(&Stack);

  Stack_Int_Push(&Stack, 5);
  Stack_Int_Push(&Stack, 4);
  Stack_Int_Push(&Stack, 3);

  Insert_At_Bottom(&Stack, 6);

  printf("Top:");
  while (!Stack_Int_Empty(&Stack))
  {
    printf("\t%d\n", Stack_Int_Pop(&Stack));
  }

  return 0;

} /* main() */
```

Complexity Analysis

- Push: O(1)
- Pop: O(1)
- Stack Top: O(1)
- Empty: O(1)
- Insert at Bottom: O(n) time complexity, O(n) space complexity (additional memory needed to perform the computation)

What is wrong?

- Pop still executes on a empty stack!
- Implementation needs to consider memory constraints
  - When Push executes on a full stack, what happens?
- How to solve these problems?

Solutions

- A not-so-elegant solution by assert(expression)
  - Abort if expression is false
- Should handle failure gracefully
  - Check if stack empty before proceeding
  - Use a global error variable
    - Set variable to error code and check it
    - Create error recovery/notification functions

stack_int.c changes for preconditions

```
int Stack_Int_Stacktop(Stack_Int_t *S_Ptr)
{
  /* view and return item at top of stack */

  int Result = 0;

  /* precondition: what must be satisfied to execute
     the operation */

  assert(!Stack_Int_Empty(S_Ptr));

  Result = S_Ptr->Items[S_Ptr->Top];

  return Result;
} /* Stack_Int_Stacktop() */
```

In plain English: the expression in assert() means "Stack not empty". Therefore, abort if stack not empty is false or abort if stack is empty

Have to include assert.h

Other ways to enforce precondition

- How about:

```
assert(    (S_Ptr != NULL)
        && (!Stack_Int_Empty(S_Ptr)));
```

- Or define the macro (in .h file)

```
#define Stack_Int_OK(SP) ((NULL != (SP)) \
  && ((SP)->Top >= -1) \
  && ((SP)->Top < STACK_INT_SIZE))
```

and use (in .c files):

```
assert(    (Stack_Int_OK(S_Ptr))
        && (!Stack_Int_Empty(S_Ptr)));
```

stack_int.h additions for error variable..

```
// error codes
#define STACK_INT_NO_ERROR  0
#define STACK_INT_OVERFLOW  -1
#define STACK_INT_UNDERFLOW -2

// global error variable
extern int Stack_Int_Errno;

// error handling functions
int Stack_Int_Error(void);
void Stack_Int_Clear_Error(void);
void Stack_Int_Print_Error(void);
```

stack_int.c additions for error variable..

```
int Stack_Int_Stacktop(Stack_Int_t *S_Ptr)
{
  /* view and return item at top of stack */

  if (Stack_Int_Empty(S_Ptr))
  {
    Stack_Int_Errno = STACK_INT_UNDERFLOW;
    return (0);
  }
  return (S_Ptr->Items[S_Ptr->Top]);

} /* Stack_Int_Stacktop() */
```

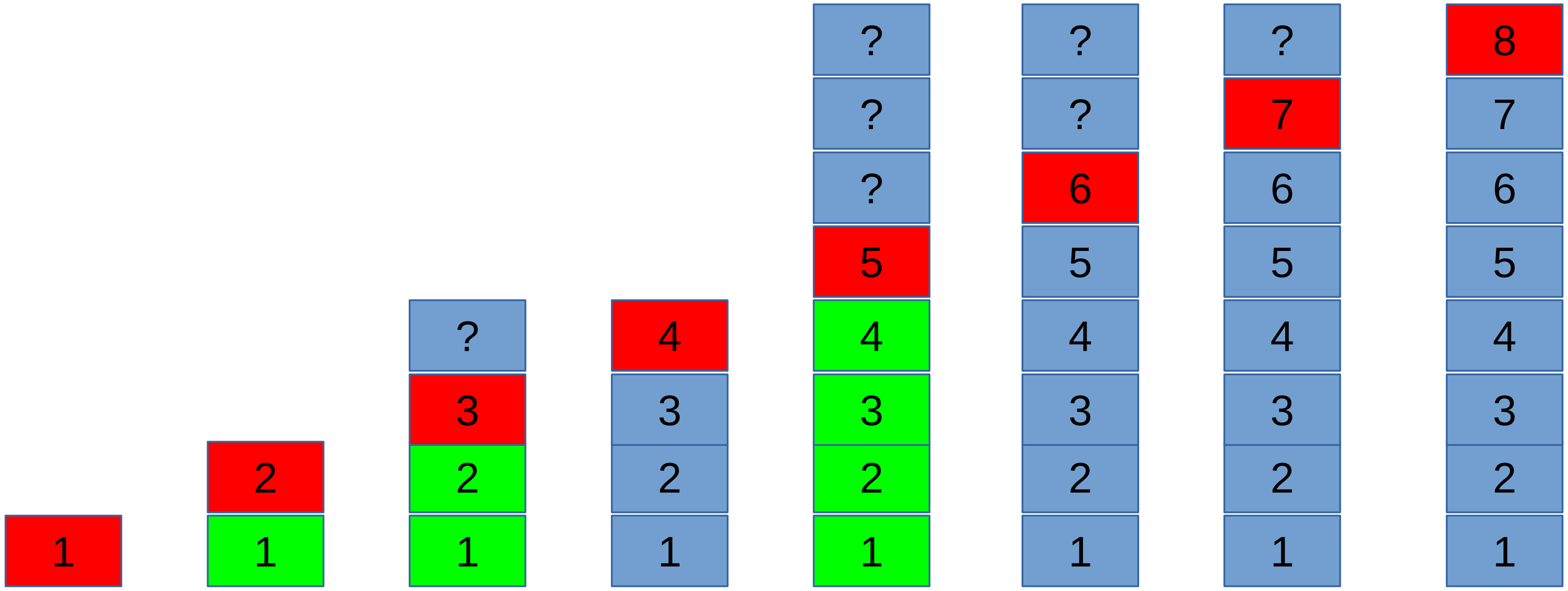
Must test `Stack_Int_Errno` at the completion of each stack function call

Stack implemented with an array of variable size

- Static memory allocation usually inappropriate for dynamically growing data structures
  - Use malloc() or calloc() to allocate memory
  - Use free() to deallocate memory
  - Use realloc() to grow/shrink allocated memory
    - If realloc succeeds, it returns the address of the new (re-sized) array at a possibly new location and all contents of original array up to min(old size, new size) are correct
      - If the re-sized array is at a new location, the old array is freed by realloc
      - Otherwise, the array is simply re-sized in-situ
    - If realloc fails, it returns NULL and the old array remains intact
    - Always store the address returned by realloc at a temporary variable and check whether the temporary variable stores a NULL address
    - If the temporary variable stores a non-NULL address, re-sizing is successful and we can safely replace the old address with the address in the temporary variable; otherwise, we handle the failure to re-size gracefully
- Always free memory after using the stack

How to Grow Stack

- Grow the size of the array by a multiplicative factor
- new size = $\lceil(1+r) * \text{original size}\rceil$, $r > 0$
- For example, $r = 1$ implies that we double the size of the array
- Assume initial stack size of 1,
- Assume it costs 1 operation to copy or push an item.



	Push(S,1)	Push(S,2)	Push(S,3)	Push(S,4)	Push(S,5)	Push(S,6)	Push(S,7)	Push(S,8)
Push								
Copy								
Total								

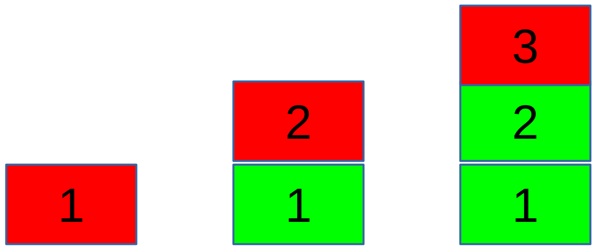
Altogether, 1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 = 15 operations
for pushing 8 items onto the stack, i.e., ~ 2 operations per push

How to Grow Stack

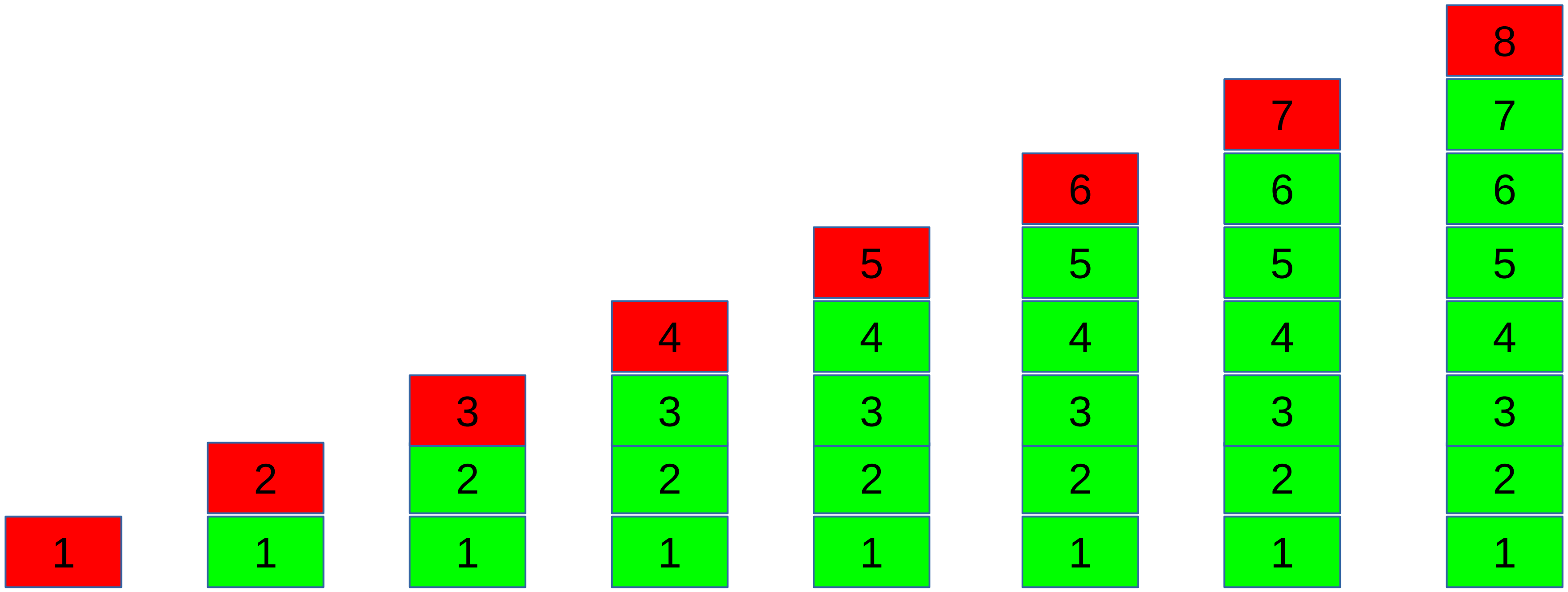
- Grow the size of the array by a multiplicative factor
- new size = $\lceil (1+r) * \text{original size} \rceil$, r > 0
- For example, r = 1 implies that we double the size of the array
- Assume initial stack size of 1, and operation cost of pushing n items (assume it costs 1 operation to copy or push an item): 1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 + 1 + 1 + 1 + …
  - Grow stack O(log n) times
  - Total complexity: Copy items O(n), push items O(n)
  - On the average, O(1) for a stack push

How (NOT) to Grow Stack

- Increase its size by a fixed amount, $F > 0$
- new size = old size + F
- Assume initial stack size of 1, and $F = 1$



	Push(S,1)	Push(S,2)	Push(S,3)	Push(S,4)	Push(S,5)	Push(S,6)	Push(S,7)	Push(S,8)
Push								
Copy								
Total								



	Push(S,1)	Push(S,2)	Push(S,3)	Push(S,4)	Push(S,5)	Push(S,6)	Push(S,7)	Push(S,8)
Push	1	1	1	1	1	1	1	1
Copy	0	1	2	3	4	5	6	7
Total	1	2	3	4	5	6	7	8

Altogether, 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36 operations
for pushing 8 items onto the stack, i.e., ~ 4 (or 8/2) operations per push

How (NOT) to Grow Stack

- Increase its size by a fixed amount, F > 0
- new size = old size + F
- Assume initial stack size of 1, and F = 1
- Operation cost of pushing n items (assume it costs 1 operation to copy or push an item): 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + …
- Total complexity: Copy items $O(n^2)$, push items $O(n)$
- Average cost of stack push is $O(n)$

When to Shrink Stack (If you have to)

- Load factor α = (# items/size of stack)
- After some pop operations, α may be low
- Assume we grow and shrink stack by the same factor
  - Shrink size by new size = old size / (1 + r)
- Shrink size of stack by a factor of (1+r) when $\alpha = 1/(1+r)^2$
- What happen when you shrink the stack by a factor of (1+r) when $\alpha = 1/(1+r)$?

C program for implementation a stack with an array of variable size

- Slides from page 30 (this page) to page 42 not covered in class
- Array is dynamically allocated within the struct Stack_ Dyn
- Initial size is 1000, and r is 0.5, multiplicative factor is (1+0.5) = 1.5

```c
/*----------------------------------------
 * program: stack_dyn.h
 * Programmer:  ece36800
 * Description:
 *   implement a dynamic stack of ints
 *----------------------------------------*/

#ifndef __stack_dyn__
#define __stack_dyn__

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define STACK_INITIAL_SIZE 1000
#define STACK_GROW_FACTOR 1.5
```

stack_dyn.h cont..

```
// define stack element type
typedef int Stack_El_t;

// declare stack structure
struct Stack_Dyn
{
  int Top;
  Stack_El_t *Items;
  int Size;
  int Minimum_Size;
};

typedef struct Stack_Dyn Stack_Dyn_t;

// function prototypes
int Stack_Dyn_Init(Stack_Dyn_t *S_Ptr, int Initial_Size);
void Stack_Dyn_Deinit(Stack_Dyn_t *S_Ptr);
Stack_Dyn_t *Stack_Dyn_New(int Initial_Size);
void Stack_Dyn_Delete(Stack_Dyn_t *S_Ptr);

int Stack_Dyn_Empty(Stack_Dyn_t *S_Ptr);
Stack_El_t Stack_Dyn_Stacktop(Stack_Dyn_t *S_Ptr);
Stack_El_t Stack_Dyn_Pop(Stack_Dyn_t *S_Ptr);
int Stack_Dyn_Push(Stack_Dyn_t *S_Ptr, Stack_El_t Item);

#endif // __stack_dyn__
```

```
/*----------------------------------------
 * program: stack_dyn.c
 * Programmer:  ece36800
 * Description: Definitions of public function
 *   Dynamic memory allocation for stack (arrays)
 *----------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack_dyn.h"


void Stack_Dyn_Deinit(Stack_Dyn_t *S_Ptr)
{
  /* deallocate memory of stack */

  free(S_Ptr->Items);
  S_Ptr->Items = NULL;
  S_Ptr->Size = 0;
  S_Ptr->Minimum_Size = 0;
  S_Ptr->Top = -1;

} /* Stack_Dyn_Deinit() */
```

stack_dyn.c cont..

```c
int Stack_Dyn_Init (
  Stack_Dyn_t *S_Ptr,
  int Initial_Size)
{
  /* initialize the stack of type El_t */

  if (Initial_Size <= 0)
    Initial_Size = STACK_INITIAL_SIZE;
  S_Ptr->Top = -1;
  S_Ptr->Size = Initial_Size;
  S_Ptr->Minimum_Size = Initial_Size;
  S_Ptr->Items = (Stack_El_t *)
             malloc(sizeof(Stack_El_t) * Initial_Size);
  if (S_Ptr->Items == NULL)
  {
    S_Ptr->Size = 0;
    return FALSE;  // out of memory!
  }
  return TRUE;

} /* Stack_Dyn_Init() */
```

stack_dyn.c cont..

```
Stack_Dyn_t *Stack_Dyn_New (int Initial_Size)
{
  /* initialize the stack of type El_t */

  Stack_Dyn_t *Result_Ptr = NULL;

  Result_Ptr = (Stack_Dyn_t *)
                   malloc(sizeof(Stack_Dyn_t));
  if (Result_Ptr == NULL)
  {
    return NULL;       // out of memory!
  }
  if (!Stack_Dyn_Init(Result_Ptr, Initial_Size))
  {
    free (Result_Ptr);
    return NULL;
  }

  return Result_Ptr;

} /* Stack_Dyn_New() */
```

stack_dyn.c cont..

```
void Stack_Dyn_Delete(Stack_Dyn_t *S_Ptr)
{
  /* delete stack and the struct from memory */

  Stack_Dyn_Deinit(S_Ptr);
  free(S_Ptr);

} /* Stack_Dyn_Delete() */


int Stack_Dyn_Empty(Stack_Dyn_t *S_Ptr)
{
  /* is stack empty??  (TRUE/FALSE) */
  return (S_Ptr->Top < 0);

} /* Stack_Dyn_Empty() */


Stack_El_t Stack_Dyn_Stacktop(Stack_Dyn_t *S_Ptr)
{
  /* view and return item at top of stack */
  return (S_Ptr->Items[S_Ptr->Top]);

} /* Stack_Dyn_Stacktop() */
```

stack_dyn.c cont..

```
Stack_El_t Stack_Dyn_Pop(Stack_Dyn_t *S_Ptr)
{
  /* pull top item off stack (adjust stack) */

  Stack_El_t Result = 0;
  int New_Size = 0;
  Stack_El_t *New_Items = NULL;

  Result = S_Ptr->Items[S_Ptr->Top];
  S_Ptr->Top--;
  if (    (S_Ptr->Size == S_Ptr->Minimum_Size)
       || ((int)((S_Ptr->Top + 1) * STACK_GROW_FACTOR
             * STACK_GROW_FACTOR) + 1) >= S_Ptr->Size)
  {
    return (Result);  // no stack adjustment
  }
  New_Size = (int)((S_Ptr->Top + 1) *
                STACK_GROW_FACTOR) + 1;
  if (New_Size <= S_Ptr->Top + 1)
  {
    return (Result);  // no stack adjustment
  }
```

stack_dyn.c cont..

```
  if (New_Size < S_Ptr->Minimum_Size)
  {
    New_Size = S_Ptr->Minimum_Size;
  }
  New_Items = (Stack_El_t *)realloc(S_Ptr->Items,
                    sizeof(Stack_El_t) * New_Size);
  if (New_Items == NULL)
  {
    return Result; // problem with memory
  }

  S_Ptr->Size = New_Size;
  S_Ptr->Items = New_Items;

  return Result; // stack adjusted correctly

} /* Stack_Dyn_Pop() */
```

stack_dyn.c cont..

```
int Stack_Dyn_Push (
  Stack_Dyn_t *S_Ptr,
  Stack_El_t Item)
{
  /* place Item on top of stack (adjust stack) */

  int New_Size = 0;
  Stack_El_t *New_Items = NULL;

  if (S_Ptr->Top >= S_Ptr->Size - 1)
  {
    New_Size = (int)(STACK_GROW_FACTOR * S_Ptr->Size)+ 1;
    if (New_Size <= S_Ptr->Size)
      return FALSE; // overflow

    New_Items = (Stack_El_t *)realloc(S_Ptr->Items,
                    sizeof(Stack_El_t) * New_Size);
    if (New_Items == NULL)
      return FALSE; // out of memory!

    S_Ptr->Size = New_Size;
    S_Ptr->Items = New_Items;
  }
  S_Ptr->Items[++S_Ptr->Top] = Item;
  return (TRUE);

} /* Stack_Dyn_Push */
```

```
/*----------------------------------------
 * program: insert_at_bottom_dyn.c
 * Programmer:  ece36800
 * Description:
 *  develop function that would insert a new item at the
 *  bottom of the stack using only the public stack functions
 *---------------------------------------*/

#include <stdio.h>
#include "stack_dyn.h"

void Insert_At_Bottom(Stack_Dyn_t *S_Ptr, Stack_El_t Item)
{
  /* insert Item at bottom of stack */

  Stack_Dyn_t *Aux_Stack; /* only pointer! */
  int Temp = 0;

  Aux_Stack = Stack_Dyn_New(0);
  while (!Stack_Dyn_Empty(S_Ptr))
  {
    Temp = Stack_Dyn_Pop(S_Ptr);
    Stack_Dyn_Push(Aux_Stack, Temp);
  }
  Stack_Dyn_Push(S_Ptr, Item);
  while (!Stack_Dyn_Empty(Aux_Stack))
  {
    Temp = Stack_Dyn_Pop(Aux_Stack);
    Stack_Dyn_Push(S_Ptr, Temp);
  }
  Stack_Dyn_Delete(Aux_Stack);

} /* Insert_At_Bottom() */
```

insert_at_bottom_dyn.c cont..

```
int main(void)
{
  Stack_Dyn_t Stack;  /* memory allocated for struct */

  Stack_Dyn_Init(&Stack, 0);

  Stack_Dyn_Push(&Stack, 5);
  Stack_Dyn_Push(&Stack, 4);
  Stack_Dyn_Push(&Stack, 3);

  Insert_At_Bottom(&Stack, 6);

  printf("Top:");
  while (!Stack_Dyn_Empty(&Stack))
  {
    printf("\t%d\n", Stack_Dyn_Pop(&Stack));
  }
  Stack_Dyn_Deinit(&Stack);

  return 0;

} /* main() */
```

Further comments

- Releasing resources is important!
  - A call to init() should have a corresponding call to deinit() [new() ↔ delete() in C++]
- Make sure all unused memory/fields are destroyed
- Used Stack_El_t for stack content type
  - Need only modify the typedef and recompile
  - How to have multiple stacks of different types?
    - Cut and paste in C
    - Typecast in C
    - Create templates in C++