Stacks and Queues

LAST	TODAY	NEXT
Abstract data types	Using library interface • Stacks • Queues	Linked lists

Data structure invariants

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______ ssa_t;

int ssa_len(ssa_t A);



ssa_t ssa_new(int size);

string ssa_get(ssa_t A, int i);

void ssa_set(ssa_t A, int i, string x);
```

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______ ssa_t;

int ssa_len(ssa_t A);



ssa_t ssa_new(int size);

string ssa_get(ssa_t A, int i);



void ssa_set(ssa_t A, int i, string x);
/*@requires 0 <= i && i < ssa_len(A); @*/;
```

Self-sorting arrays: Client Perspective

```
/******************************* Interface *******************************/
// typedef ______ ssa_t;

int ssa_len(ssa_t A)
/*@ensures \result >= 0; @*/;
```


```
ssa_t ssa_new(int size)



string ssa_get(ssa_t A, int i)



void ssa_set(ssa_t A, int i, string x)
/*@requires 0 <= i && i < ssa_len(A); @*/;
```

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@ensures \result >= 0; @*/;

ssa_t ssa_new(int size)



string ssa_get(ssa_t A, int i)



void ssa_set(ssa_t A, int i, string x)
/*@requires 0 <= i && i < ssa_len(A); @*/;
```

Self-sorting arrays: Interface

```
/******************************* Interface *******************************/
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;   */
/*@ensures \result >= 0; */;

ssa_t ssa_new(int size)
```

```
string ssa_get(ssa_t A, int i)
/*@requires A != NULL;*/


void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;                    */
/*@requires 0 <= i && i < ssa_len(A); */;
```

Self-sorting arrays: Interface

```
/*************************** Interface ***************************/
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;   @*/
/*@ensures \result >= 0; @*/;

ssa_t ssa_new(int size)
/*@requires 0 <= size;                  @*/
/*@ensures \result != NULL;             @*/
/*@ensures ssa_len(\result) == size; @*/;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;                   @*/
/*@requires 0 <= i && i < ssa_len(A); @*/;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;                   @*/
/*@requires 0 <= i && i < ssa_len(A); @*/;
```

Self-sorting arrays: Library Perspective

```
// Implementation-side type

struct ssa_header {
   string[] data;
   int length;     // = \length(data)
};

typedef struct ssa_header ssa;

// Client type
typedef ssa* ssa_t;
```

connection between the type exported to the user `ssa_t` by the interface and the type `ssa`

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A != NULL;
//@ensures \result >= 0;
{
  return A->length;
}


string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}
```

SAFE?

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@ensures \result == \length(A->data);
{
 return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);

{
 return A->data[i];
}
```

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@ensures \result == \length(A->data);
{
 return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);

{
 return A->data[i];
}
```

SAFE?

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A!= NULL;
//@requires A->length = \length(A->data);
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
//@requires A->length == \length(A->data);
{
  return A->data[i];
}
```

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
int ssa_len(ssa* A)
//@requires A! == NULL;
//@requires A -> length = \length(A->data);
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires A != NULL;
//@requires 0 <= i && i < ssa_len(A);
//@requires A->length == \length(A->data);
{
  return A->data[i];
}
```

Every function that takes `ssa` as a parameter needs the same preconditions!

`ssa* A` is well-formed if

- A is a non-NULL pointer
- `\length(A) == \length(A->data)`
- A is sorted

Data structure invariants (representation invariants)

- A is a non-NULL pointer
- length of `A == \length(A->data)`
- A is sorted

```
bool is_array_expected_length(string[] A, int length) {
//@assert \length(A) == length;
  return true;
}
```

```
 // Representation invariant
 bool is_ssa(ssa* A) {
   return A != NULL
        && is_array_expected_length(A->data, A->length)
        && is_sorted(A);
 }
```

Data structure invariants (representation invariants)

```
// Representation invariant
bool is_ssa(ssa* A) {
  return A != NULL
      && is_array_expected_length(A->data, A->length)
      && is_sorted(A);
}

int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}
```

- Every function that takes a self-sorting array `A` as input must have `//@requires is_ssa(A);`

- Every function that modifies a self-sorting array `A` must have `//@ensures is_ssa(A);`

- Every function that returns a self-sorting array `A` must have `//@ensures is_ssa(A);`

Self-sorting arrays: Implementation

```
struct ssa_header {
  string[] data;
  int length;
};
typedef struct ssa_header ssa;
```

```
ssa* ssa_new(int size)
//@requires 0 <= size;
//@ensures is_ssa(\result);
//@ensures ssa_len(\result) == size;
{
  ssa* A = alloc(ssa);
  A->data = alloc_array(string, size);
  A->length = size;
  return A;
}


// Client type
typedef ssa* ssa_t;
```

Stacks and Queues

The stack interface

```
//typedef ______* stack_t;

bool stack empty(stack t S) // O(1)
//@requires S != NULL;
;
```

```
stack_t stack_new() // O(1)
//@ensures \result != NULL;
//@ensures stack_empty(\result);
;

void push(stack_t S, string x) //O(1)
//@requires S != NULL;


string pop(stack_t S) // O(1)
//@requires S!= NULL;
//@requires !stack_empty(S);
```

Last in First Out (LIFO) structure

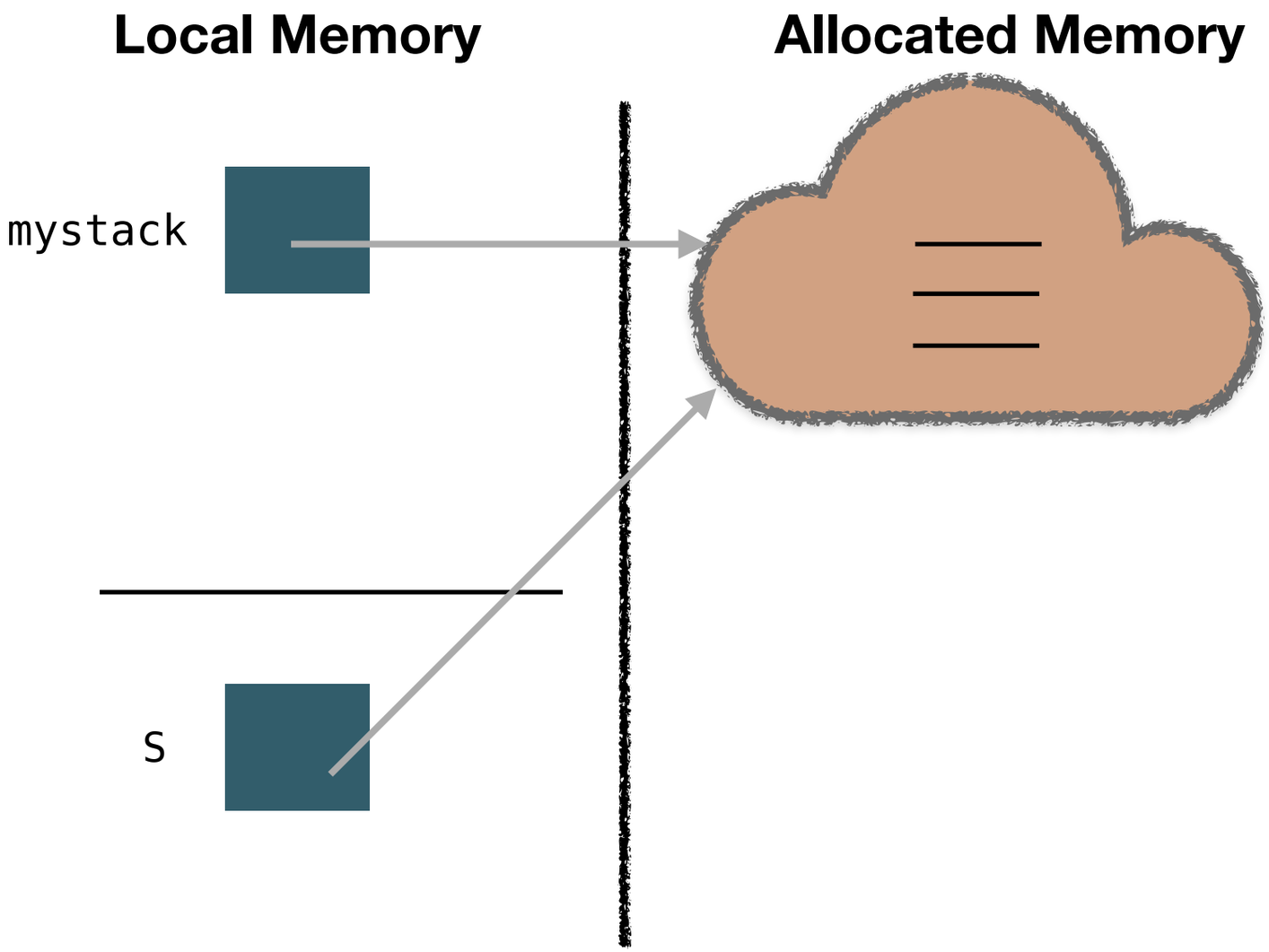
Example: peek

```
bool stack_empty(stack_t S) // O(1)
//@requires S != NULL;
;

stack_t stack_new(stack* S) // O(1)
//@ensures \result != NULL;
//@ensures stack_empty(\result);
  ;

void push(stack_t S, string x) //O(1)
//@requires S != NULL;
;

string pop(stack_t S) // O(1)
//@requires S != NULL;
//@requires !stack_empty(S);
```

Write a function that returns the top element without removing it,
using only the functions from the stack interface.

```
string peek(stack_t S)
//requires S != NULL && !stack_empty(S);

{
  string x = pop(S);
  push(S,x);
  return x;
}
```

Client (caller) view




```
string peek(stack_t S)
//@requires S != NULL &&
stack_empty(S);
{
  string s = pop(S);
  push(S,s);
  return s;
}

//Client code
stack_t mystack = stack_new();
push(mystack, "A");
push(mystack, "B");
string s = peek(mystack);
```

Example: `size` (first attempt)

```
bool stack_empty(stack_t S) // O(1)
//@requires S != NULL;
;

stack_t stack_new(stack* S) // O(1)
//@ensures \result != NULL;
//@ensures stack_empty(\result);
  ;

void push(stack_t S, string x) //O(1)
//@requires S != NULL;
;

string pop(stack_t S) // O(1)
//@requires S != NULL;
//@requires !stack_empty(S);
```

Write a function that returns the number of elements in a stack,
using only the functions from the stack interface

```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  int count = 0;
  while (!stack_empty(S)){
    pop(S);
    count++;
  }
  return count;
}
```

What is undesirable about this solution?

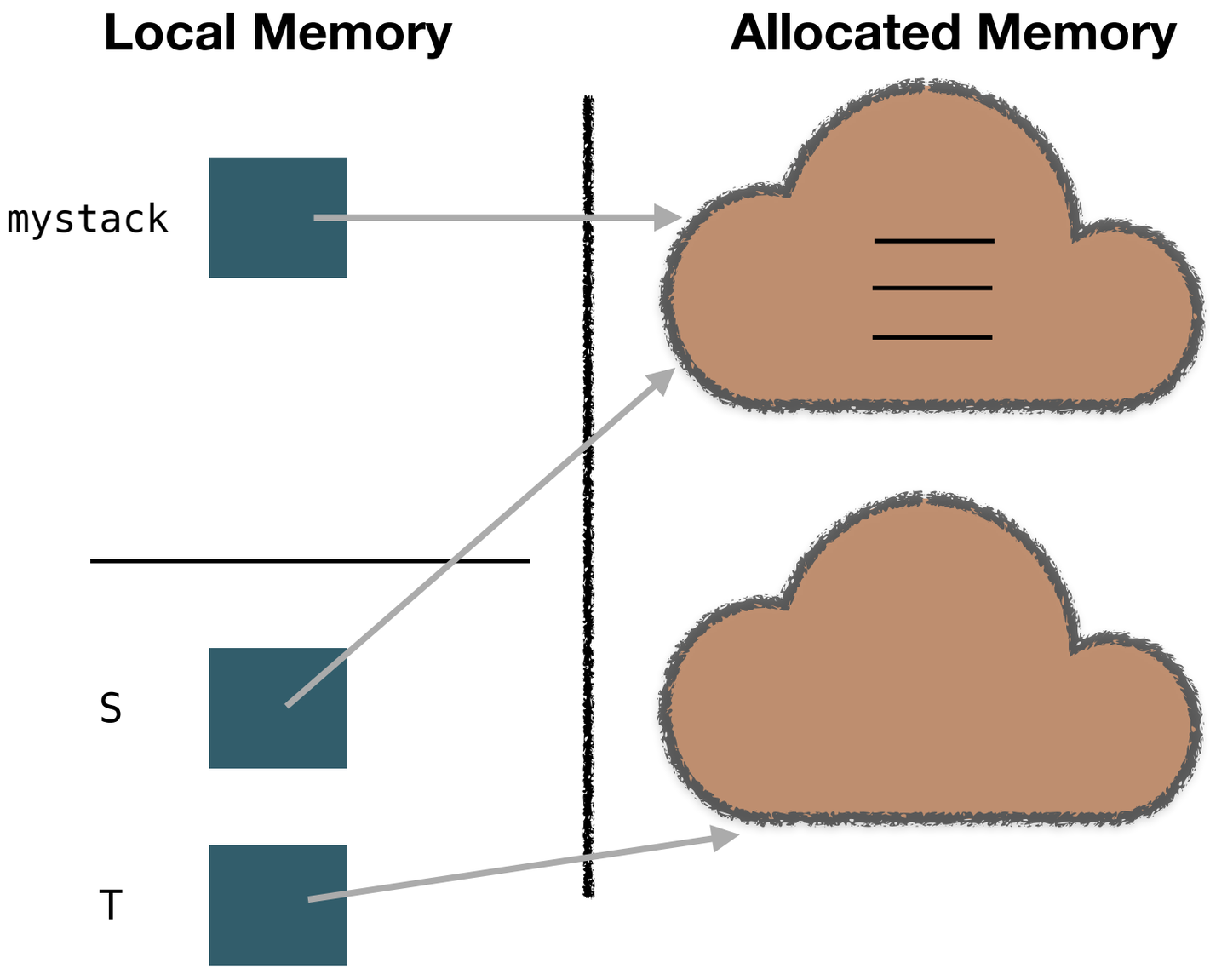
Example: `size` (second attempt)

Write a function that returns the number of elements in a stack,
using only the functions from the stack interface

```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)){
    string x = pop(S);
    push(T, s);
    count++;
  }
  //@assert stack_empty(S);
  S = T;
  return count;
}
```

What is undesirable about this solution?

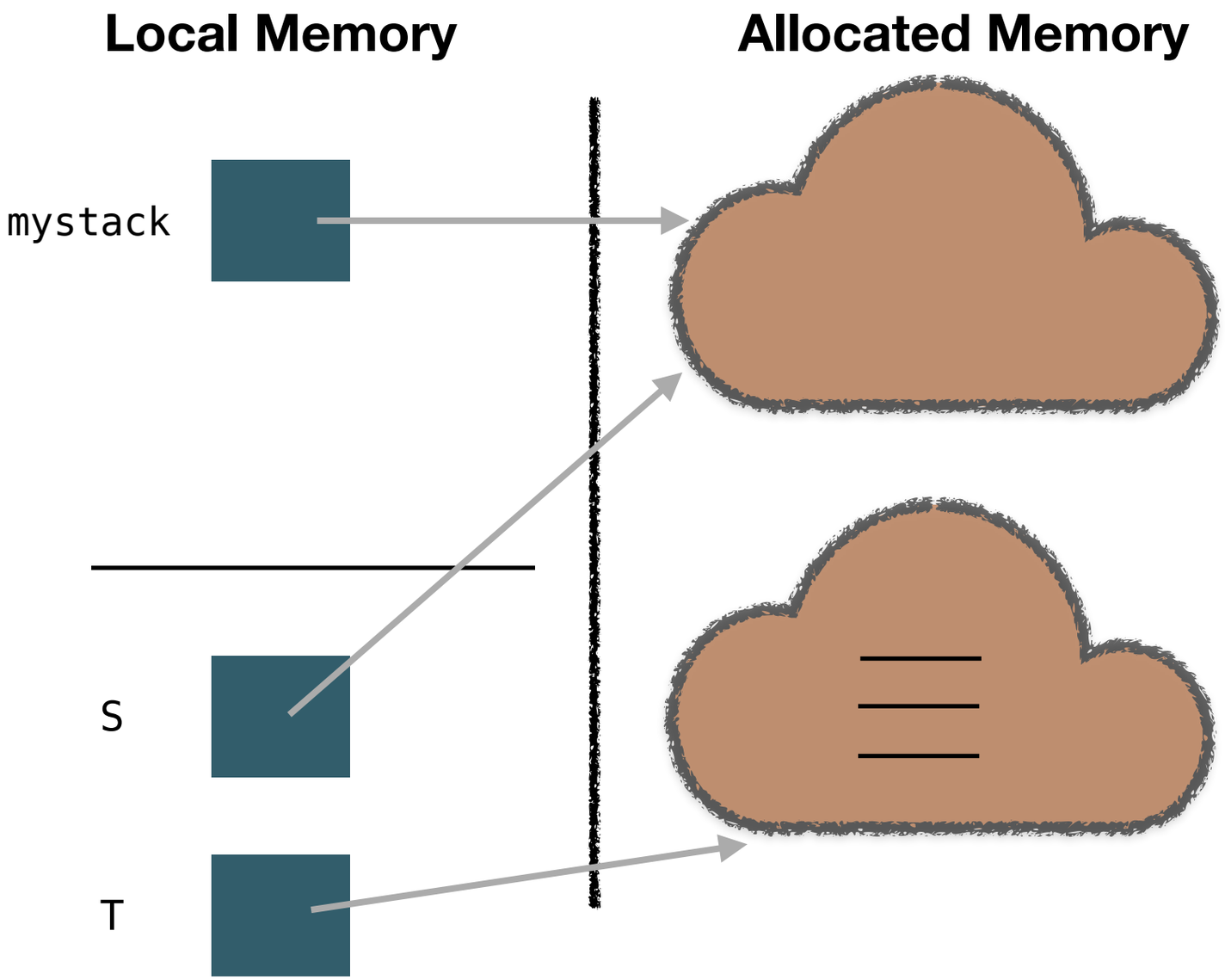
Client (caller) view




```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)){
    string x = pop(S);
    push(T, s);
    count++;
  }
  //@assert stack_empty(S);
  S = T;
  return count;


//Client code
stack_t mystack = stack_new();

….

int s = size(mystack);
```

After the the function **`size`** creates a new stack T

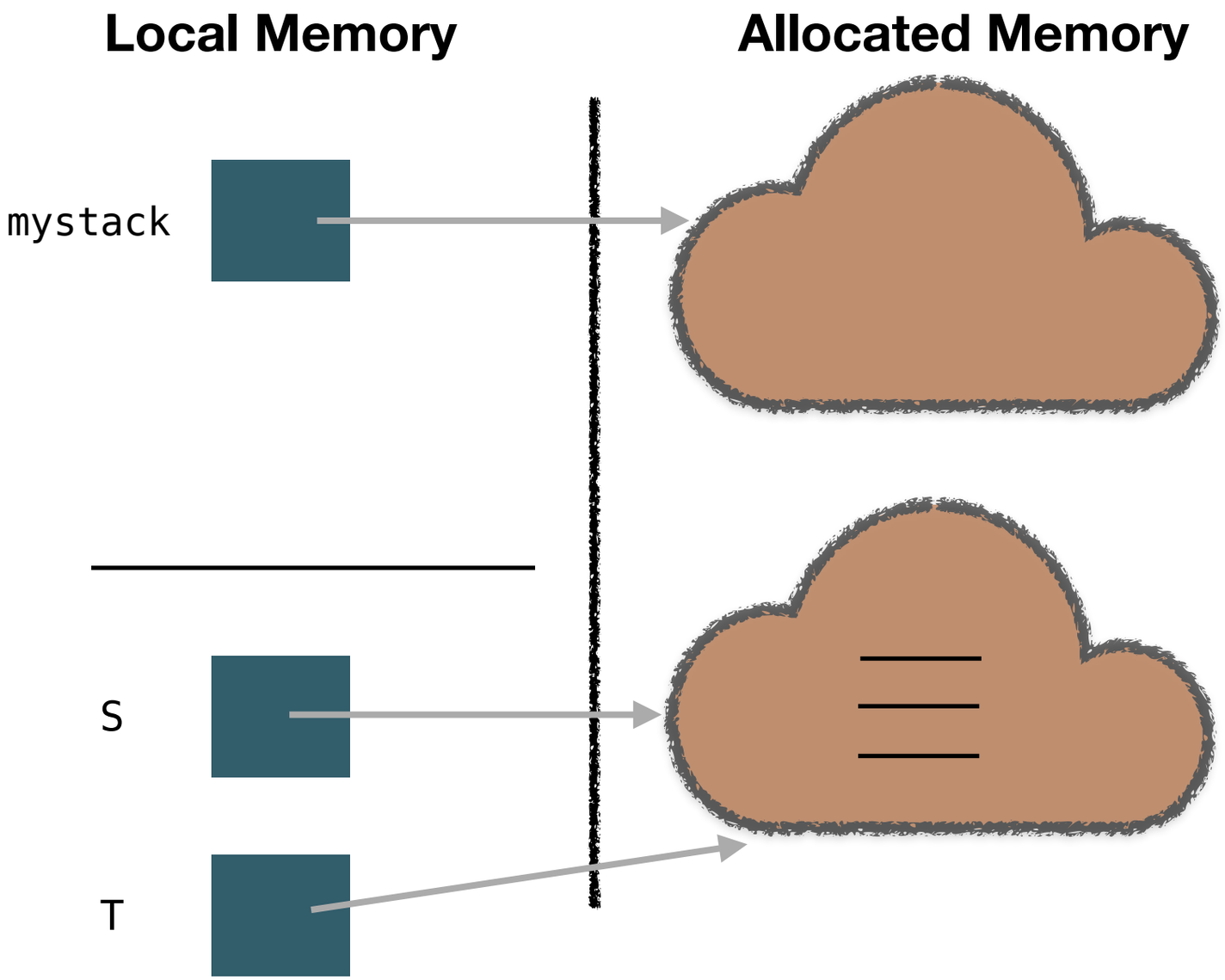
Client (caller) view




```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)){
    string x = pop(S);
    push(T, s);
    count++;
  }
  //@assert stack_empty(S);
  S = T;
  return count;


//Client code
stack_t mystack = stack_new();

….

int s = size(mystack);
```

After the the function **`size`** finishes the loop

Client (caller) view




```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)){
    string x = pop(S);
    push(T, s);
    count++;
  }
  //@assert stack_empty(S);
  S = T;
  return count;



//Client code
stack_t mystack = stack_new();

….

int s = size(mystack);
```

After the the function **`size`** sets S to T

Example: `size` (third attempt)

```
int size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)){
    string x = pop(S);
    push(T, s);
    count++;
  }
//@assert stack_empty(S);
  while (!stack_empty(T){
    push(S, pop(T)
  }
  //@assert stack_empty(T);/
  return count;
}
```

O(n)

First In First Out (FIFO)

The queue interface

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)          /* O(1) */
/*@requires Q != NULL; @*/;

queue_t queue_new()                  /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)        /* O(1) */
/*@requires Q != NULL; @*/;

string deq(queue_t Q)                /* O(1) */
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

Example: `queue_copy` (first attempt)

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)        /* O(1) */
  /*@requires Q != NULL; @*/;

queue_t queue_new()                /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)      /* O(1) */
  /*@requires Q != NULL; @*/;

string deq(queue_t Q)              /* O(1) */
  /*@requires Q != NULL; @*/
  /*@requires !queue_empty(Q); @*/ ;
```

Write a function that returns a (deep) copy of a queue, **using only the functions from the queue interface.**

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
//@ensures \result != NULL;
{
 queue_t C = Q;
 return C;
}
```

What is undesirable about this solution?

Example: `queue_copy` (second attempt)

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
//@ensures \result != NULL;
{
  queue_t C = queue_new();
  while (!queue_empty(Q)){
    string x = deq(Q);
    enq(C,x);
  }
  return C;
}
```

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)       /* O(1) */
  /*@requires Q != NULL; @*/;

queue_t queue_new()               /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)     /* O(1) */
  /*@requires Q != NULL; @*/;

string deq(queue_t Q)             /* O(1) */
  /*@requires Q != NULL; @*/
  /*@requires !queue_empty(Q); @*/ ;
```

What is undesirable about this solution?

Example: `queue_copy` (third attempt)

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
//@ensures \result != NULL;
{
  queue_t C = queue_new();
  while (!queue_empty(Q)){
    string x = deq(Q);
    enq(C,x);
    enq(Q,x);
  }
  return C;
}
```

What is wrong with this solution?

Example: `queue_copy` (fourth attempt)

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
//@ensures \result != NULL;
{
  queue_t C = queue_new();
  queue_t T = queue_new();
  while (!queue_empty(Q)){
    string x = deq(Q);
    enq(C,x);
    enq(T,x);
  }
  Q = T;
  return C;
}
```

Example: `queue_copy` (fifth attempt)

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
//@ensures \result != NULL;
{
  queue_t C = queue_new();
  queue_t T = queue_new();
  while (!queue_empty(Q)){
    string x = deq(Q);
    enq(C,x);
    enq(T,x);
  }
  //@assert queue_empty(Q);

  while (!empty_queue(T)
  {
    enq(Q, deq(T));
  }
  return C;
}
```