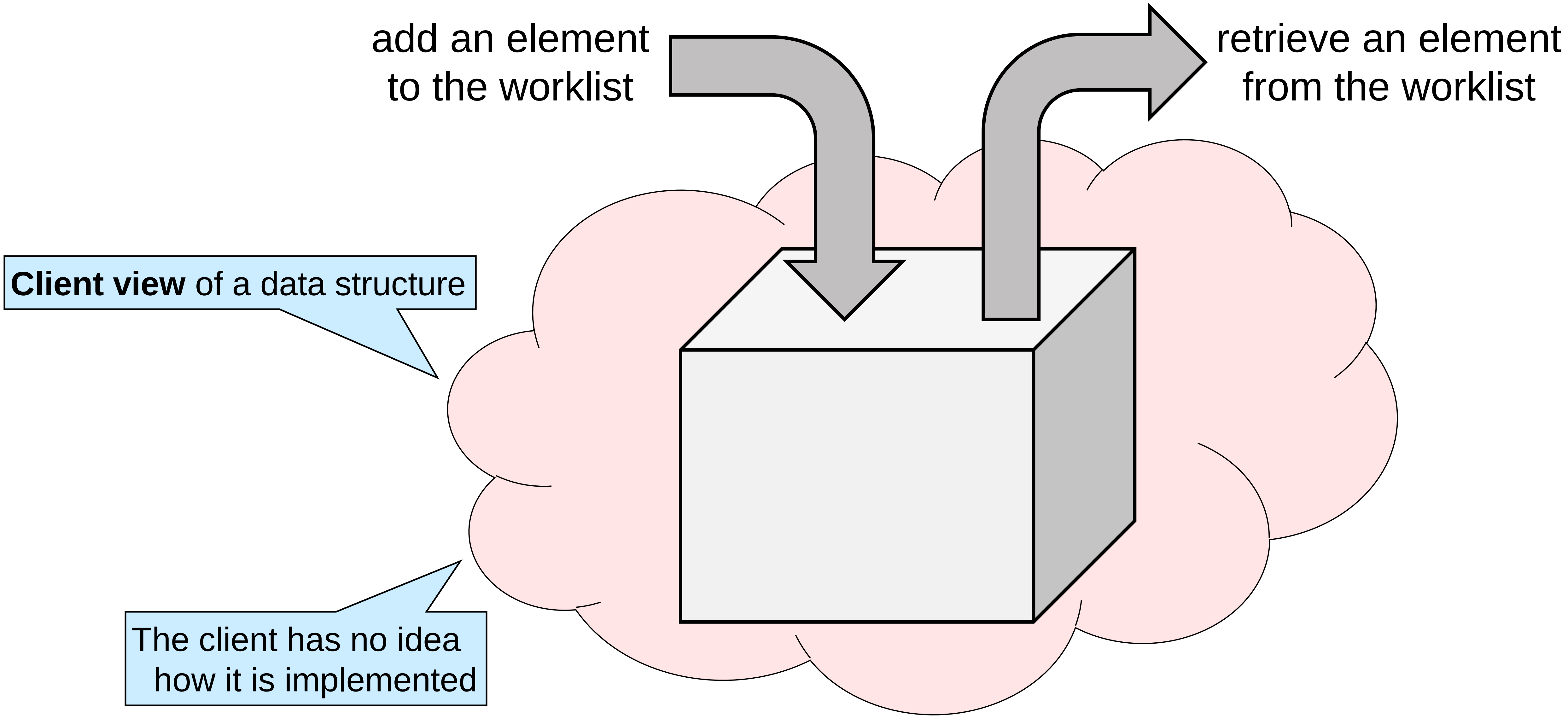
Stacks and Queues

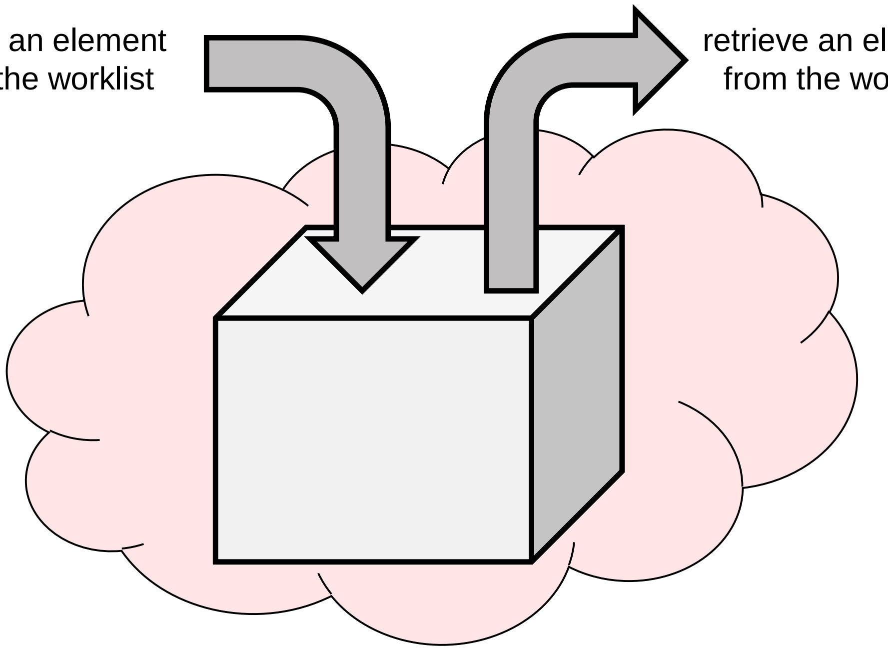
Worklists

Worklists

- A family of data structures that
  - can hold elements and
  - give us a way to get them back

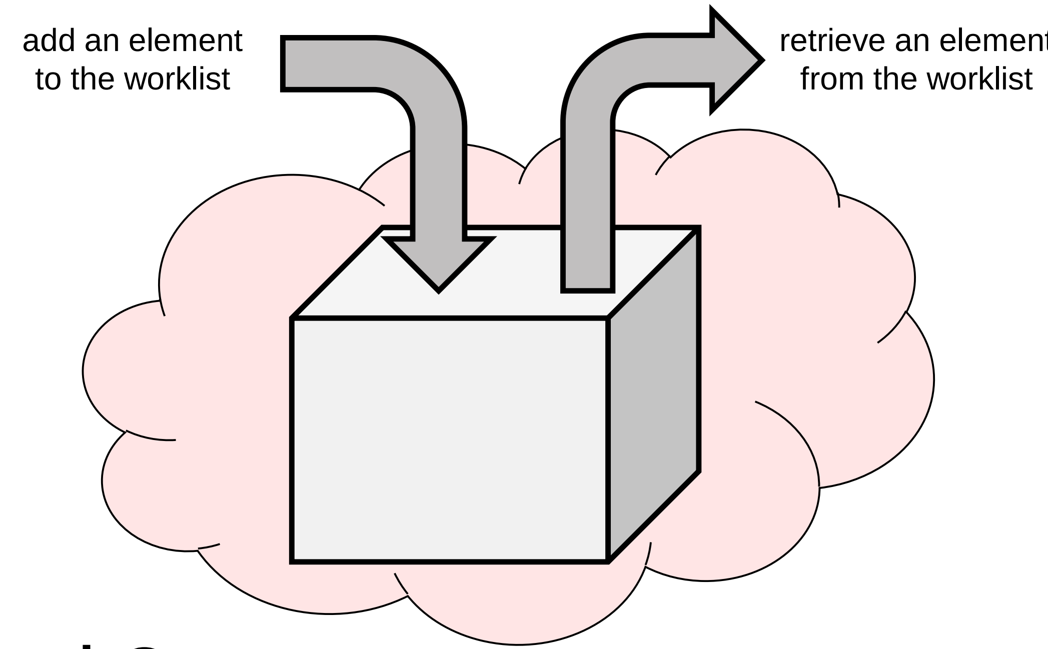


Worklists



- A family of data structures that
  - can hold elements and
  - give us a way to get them back
- Examples
  - to-do list
  - cafeteria line
  - suspended processes in an OS, …
- Pervasively used in computer science
  - This will be our first "real" data structures

Concrete Worklists



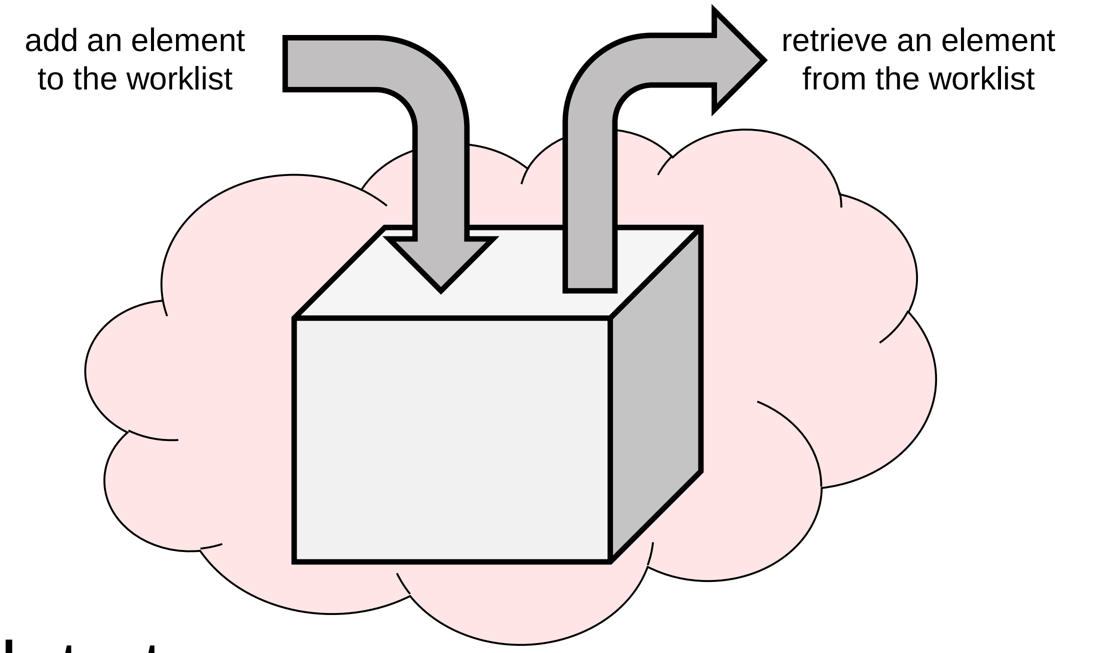
- Adding an element simply puts it in the worklist
- But which element should we get back?
  - ➢ Several options
  - o **Stacks**: retrieve the element inserted most recently
    - ➢ The LIFO data structure

      Last
      In
      First
      Out
  - o **Queues**: retrieve the element that has been there longest
    - ➢ The FIFO data structure

      First
      In
      First
      Out
  - o **Priority** queues: retrieve the most “interesting” element

We will talk about them later on

The Worklist Interface



- Turn the idea of a worklist into a data structure
  - Develop an **interface** for an abstract data type

- Types
  - Elements in the worklist: string
  - Worklist itself: wl_t

We will generalize this later on

This is the abstract type of worklists

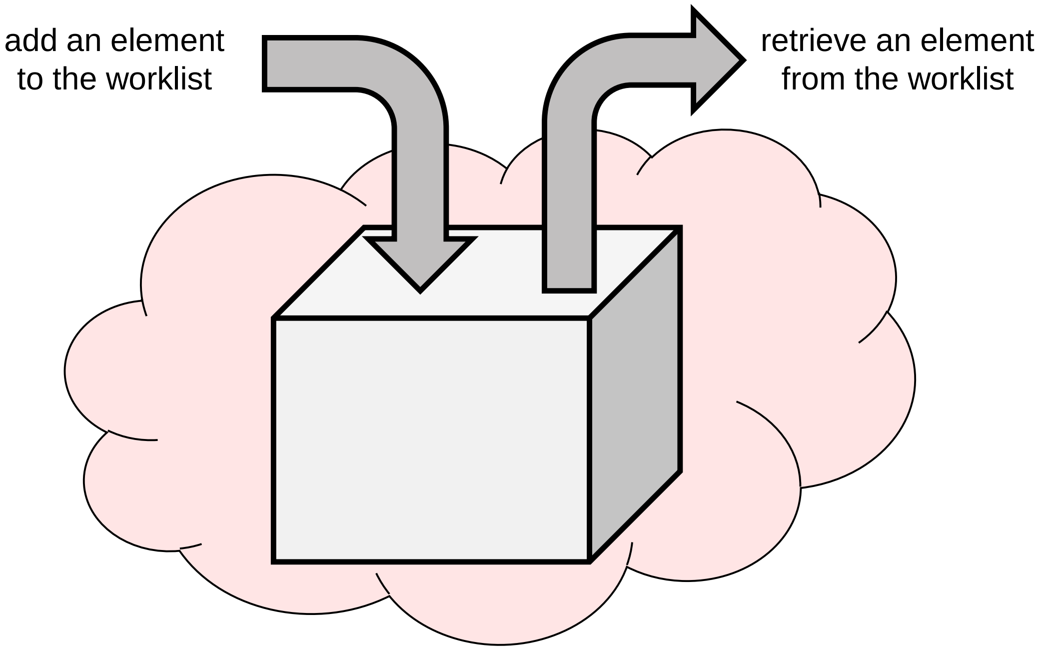
A pointer type

- Operations
  - add an element: wl_add
  - retrieve an element: wl_retrieve
  - create a new worklist: wl_new
  - check if the worklist is empty: wl_empty
    - we cannot retrieve anything from an empty worklist!

There is **no** wl_full.
We are considering **unbounded worklists**

can hold arbitrarily many elements

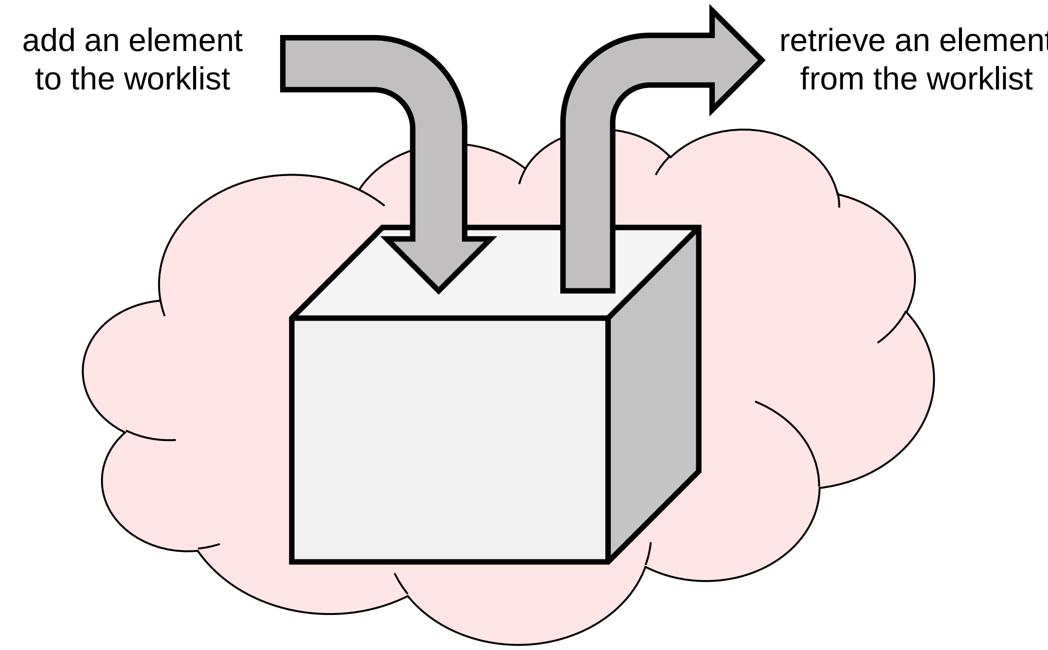
Worklist Interface



- Operands and contracts
  - add an element: wl_add
    - Takes in a worklist and an element
    - Worklist is not empty as a result
  - retrieve an element: wl_retrieve
    - Takes in a worklist, returns an element
    - Worklist must not be empty
  - create a new worklist: wl_new
    - Takes in nothing, returns an empty worklist
  - check if the worklist is empty: wl_empty
    - Takes in a worklist, returns a boolean

+ a bunch of NULL-checks

The Worklist Interface



Worklist Interface

```
// typedef ______* wl_t;

bool wl_empty(wl_t W)
  /*@requires W != NULL;          @*/ ;

wl_t wl_new()
  /*@ensures \result != NULL;     @*/
  /*@ensures wl_empty(\result);  @*/ ;

void wl_add(wl_t W, string x)
  /*@requires W != NULL;          @*/
  /*@ensures !wl_empty(W);        @*/ ;

string wl_retrieve(wl_t W)
  /*@requires W != NULL;          @*/
  /*@requires !wl_empty(W);       @*/ ;
```

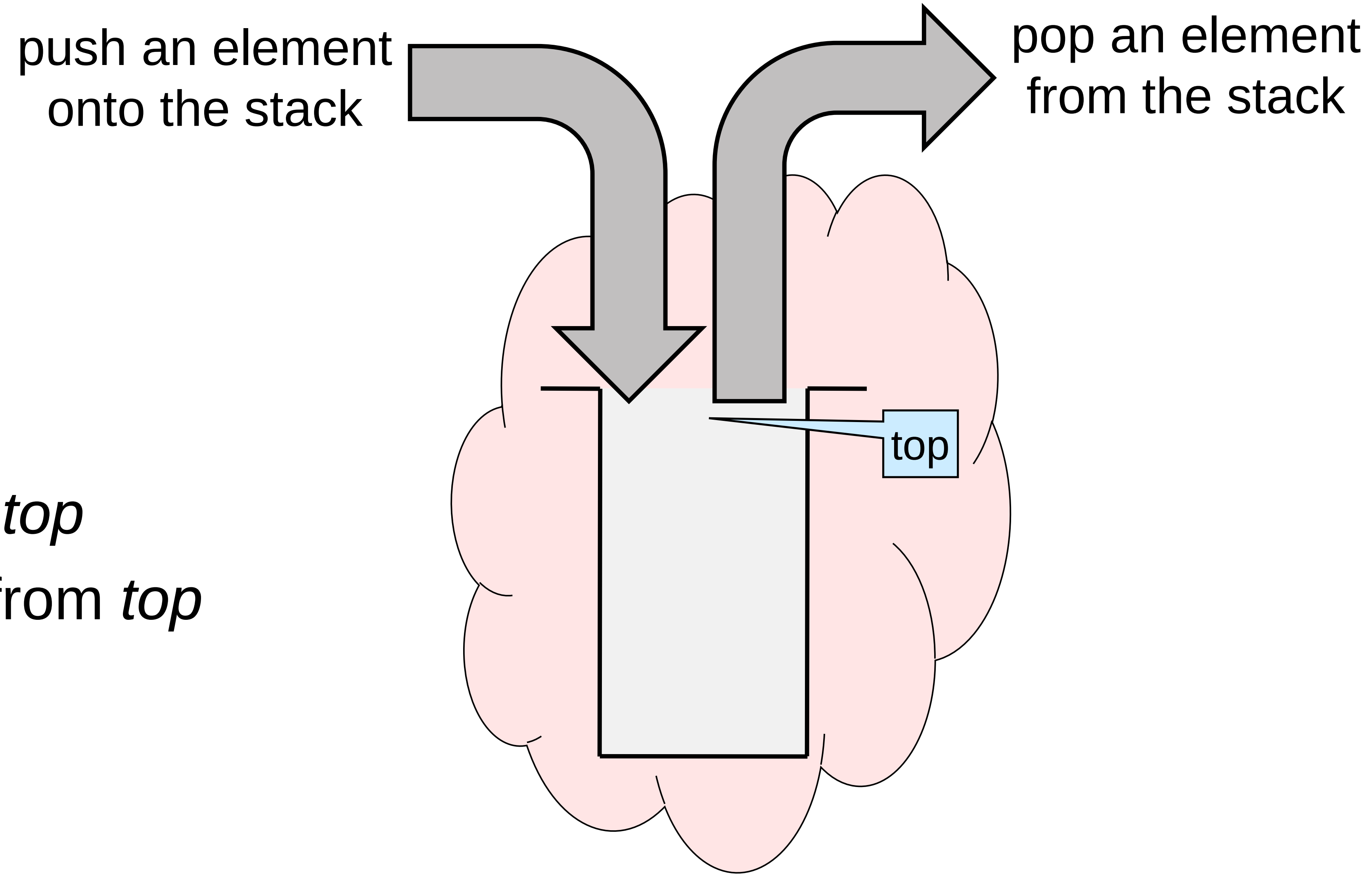
What

- This will be a **template** for the concrete worklists we will be working with
  - stacks and queues
  - We will never use this interface
  - We will use instances for stacks and for queues

Stacks

Stacks

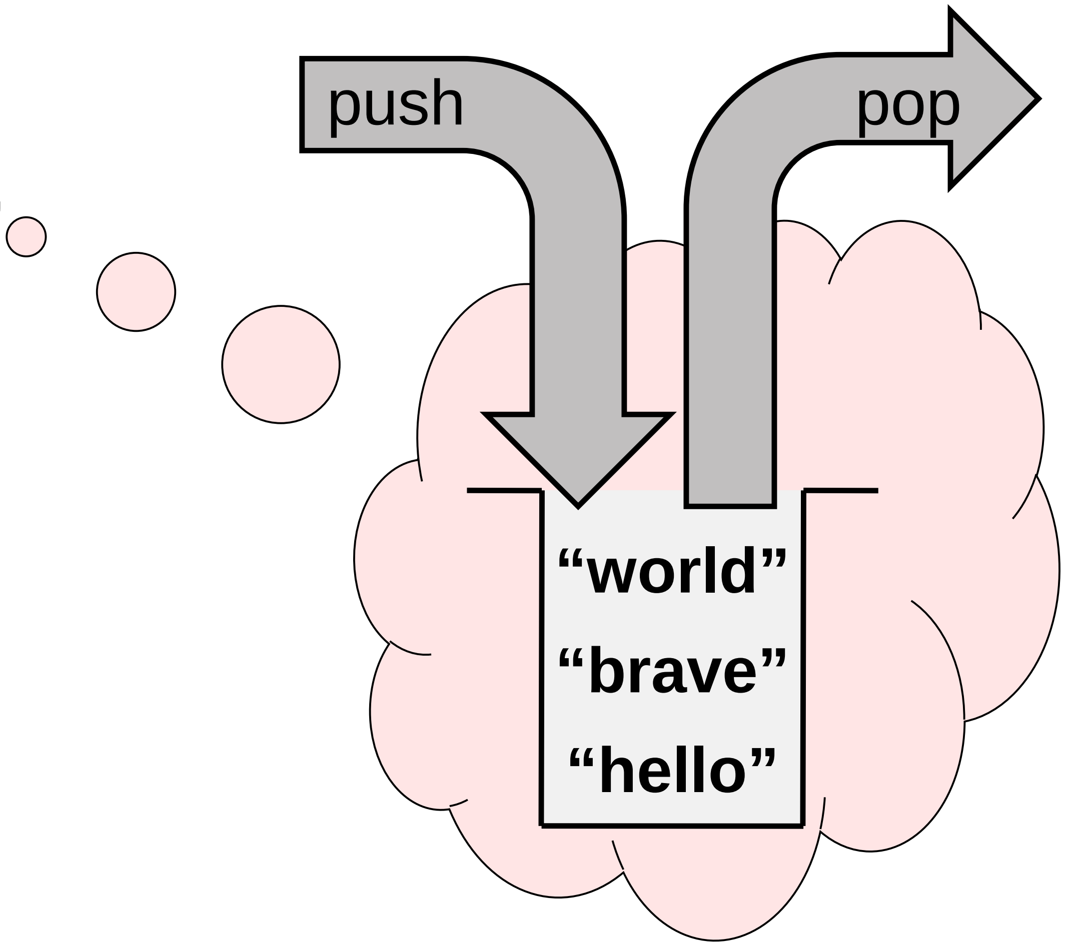
- A worklist where we retrieve the last inserted element
  - **F**irst **I**n **L**ast **O**ut
  - Like a stack of books



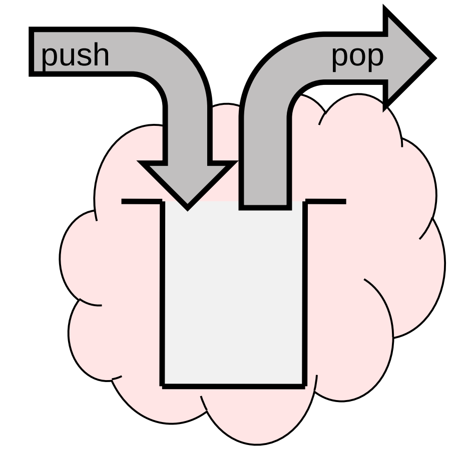
- Traditional name of operations
  - **push** (= add) on *top*
  - **pop** (= retrieve) from *top*

Stacks

- A worklist where we pop the last element pushed
  - **F**irst **I**n **L**ast **O**ut
- If we push
  - **"hello"** then **"brave"** then **"world"**
- and then pop, we get
  - **"world"**
- and then pop again, we get
  - **"brave"**
- and pop once more, we get
  - **"hello"**
- at this point the stack is empty



The Stack Interface



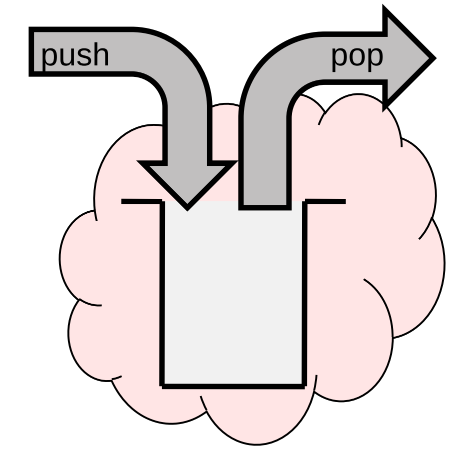
Stack Interface

```
// 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;             @*/
  /*@ensures !stack_empty(S);        @*/ ;

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

What

- This is the worklist interface with the names changed
- We are providing **complexity bounds** in the interface
  - We promise the stack library will implement the operations to have these cost
    - all stack operations have constant cost

The Stack Interface



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;             @*/
  /*@ensures !stack_empty(S);        @*/ ;

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

What

- Since stacks implement a **L**ast **I**n **F**irst **O**ut policy, what about adding

  //@ensures(string_equal(pop(S), x);

  as a postcondition to push?

- pop(S) changes S!
  - Running with and without contracts enabled could produce different outcomes
  - This contract is not **pure**
  - The C0 compiler has a **purity check** that catches this

Using only functions from the stack interface

Peeking into a Stack

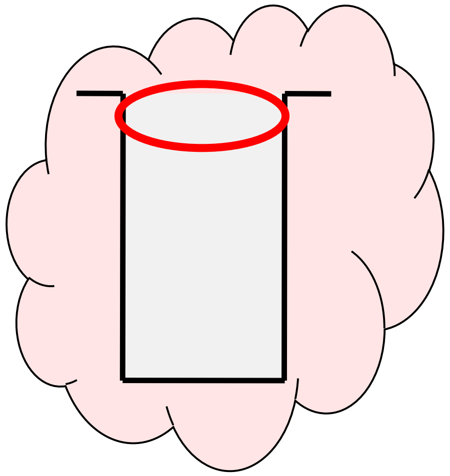
Write a **client** function that returns the top element of the stack *without removing it*

- We can do that only if the stack is not empty
  - This is a precondition
- Simply pop the stack in a variable, push the element back, and return the value of the variable

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;          @*/
  /*@ensures !stack_empty(S);     @*/ ;

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

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



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;           @*/
  /*@ensures !stack_empty(S);      @*/ ;

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

Peeking into a Stack

Write a **client** function that returns the top element of the stack *without removing it*

```
1. string peek(stack_t S)
2. //@requires S != NULL;
3. //@requires !stack_empty(S);
4. {
5.   string x = pop(S);
6.   push(S, x);
7.   return x;
8. }
```

- Is this code safe?
  - o stack_empty(S):
    - ➢ S != NULL by line 2
  - o pop(S):
    - ➢ S != NULL by line 2
    - ➢ !stack_empty(S) by line 3
  - o push(S, x)
    - ➢ S != NULL by line 2

Peeking into a Stack

Write a **client** function that returns the top element of the stack *without removing it*

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

- What is the asymptotic complexity?
  - pop(S): O(1)
  - push(S, x): O(1)
  - return x O(1)

  Total: **O(1)**

Complexity guarantees in the interface allow us to determine the cost of client functions

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;            @*/
  /*@ensures !stack_empty(S);       @*/ ;

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

Using only functions from the stack interface

Peeking into a Stack

Write a **client** function that returns the top element of the stack *without removing it*

- What about *this* implementation?

```
string peek(stack_t S)
//@requires S != NULL;
//@requires !stack_empty(S);
{
  return S->data[S->top];
}
```

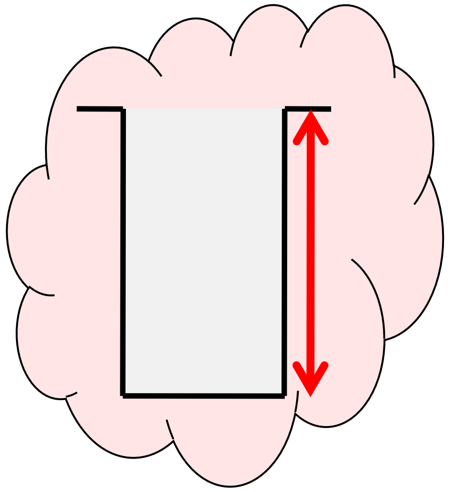
  - It assumes stacks are implemented as structs with a *data* and a *top* field
    - but we don’t know anything about how stacks are implemented!
    - all we have is an interface
  - This **violates the interface** of the stack library ✘

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;          @*/
  /*@ensures !stack_empty(S);     @*/ ;

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

The Size of a Stack

Write a **client** function that returns the number of elements in a stack



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;              @*/
  /*@ensures !stack_empty(S);         @*/ ;

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

Using only functions from the stack interface

The Size of a Stack

Write a **client** function that returns the number of elements in a stack

- count the elements as we pop them

Stack Interface

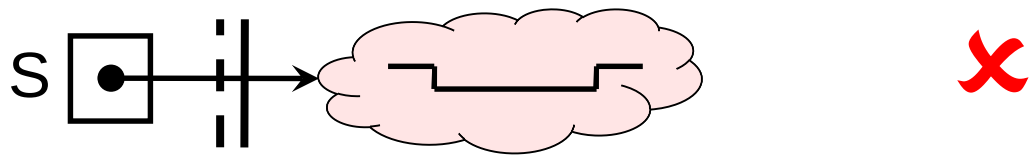
```
// typedef ______* stack_t;

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;            @*/
  /*@ensures !stack_empty(S);       @*/ ;

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

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

v.1

Exercise:
check that this code is safe

- Does this do what we want?
  - It returns the number of elements S *started with* …
  - … but S has been **emptied out** by the time we return!



- *Idea:*
  - Save the contents of S somewhere …
  - … in another stack

The Size of a Stack

Write a **client** function that returns the number of elements in a stack

- save the elements of S in another stack

```
int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  int c = 0;
  stack_t TMP = stack_new();    // ADDED
  while (!stack_empty(S)) {
    string x = pop(S);          // MODIFIED
    push(TMP, x);               // ADDED
    c++;
  }
  //@assert stack_empty(S);     // ADDED
  S = TMP;                      // ADDED
  return c;
}
```

v.2

Exercise:
check that this code is safe

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;          @*/
  /*@ensures !stack_empty(S);     @*/ ;

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

- Does this do what we want?
  - TMP is in reverse order
    - so S is in reverse order at the end

✘

  - On return, **the caller stack is empty**
    - What??

✘

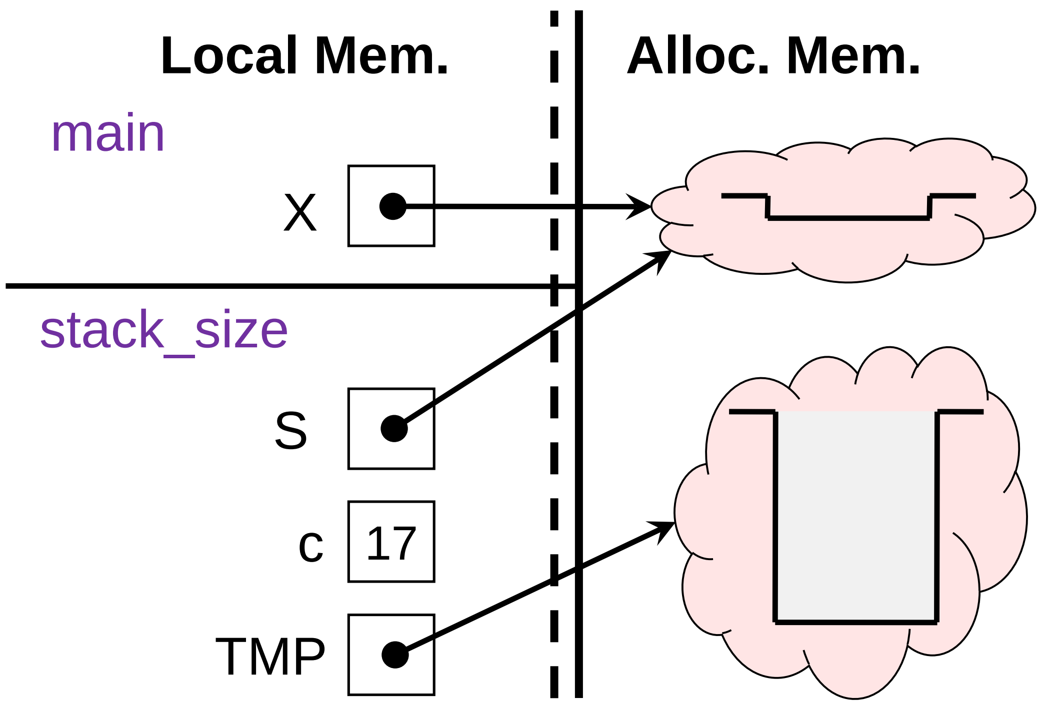
The Size of a Stack

- On return, the caller stack is empty

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

v.2

```
int main() {
  …
  stack_t  X = stack_new();
  …
  … stack_size(X)
  …
  return 0;
}
```

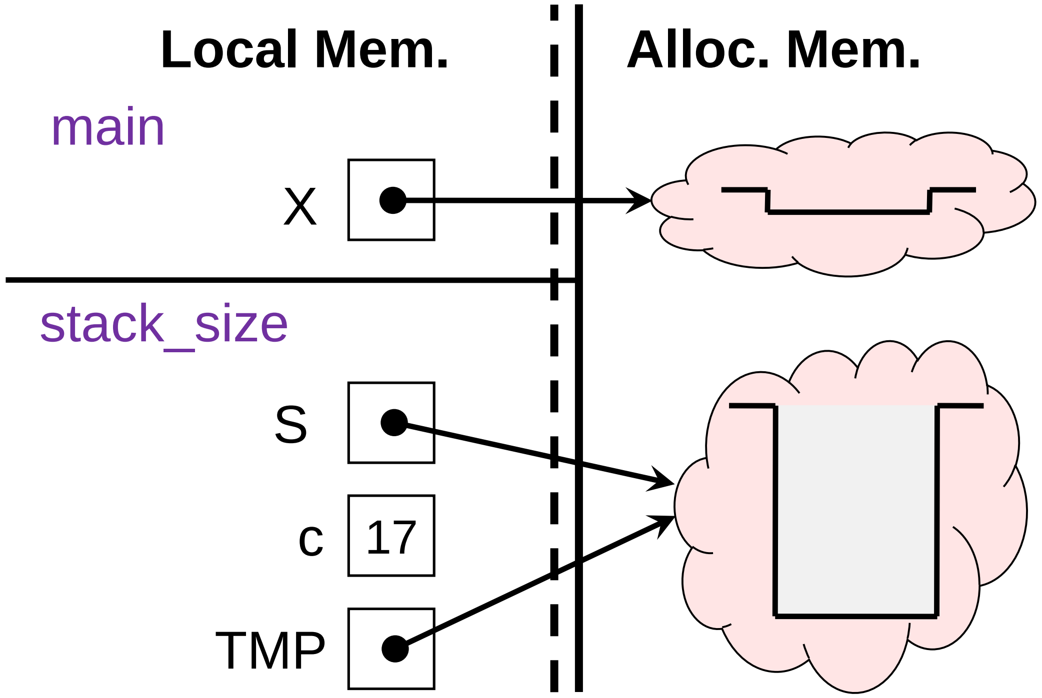


The Size of a Stack

- On return, the caller stack is empty

```
int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  int c = 0;
  stack_t TMP = stack_new();
  while (!stack_empty(S)) {
    string x = pop(S);
    push(TMP, x);
    c++;
  }
  //@assert stack_empty(S);
  S = TMP;
  return c;
}                                   v.2
```

```
int main() {
  …
  stack_t  X = stack_new();
  …
  … stack_size(X)
  …
  return 0;
}
```



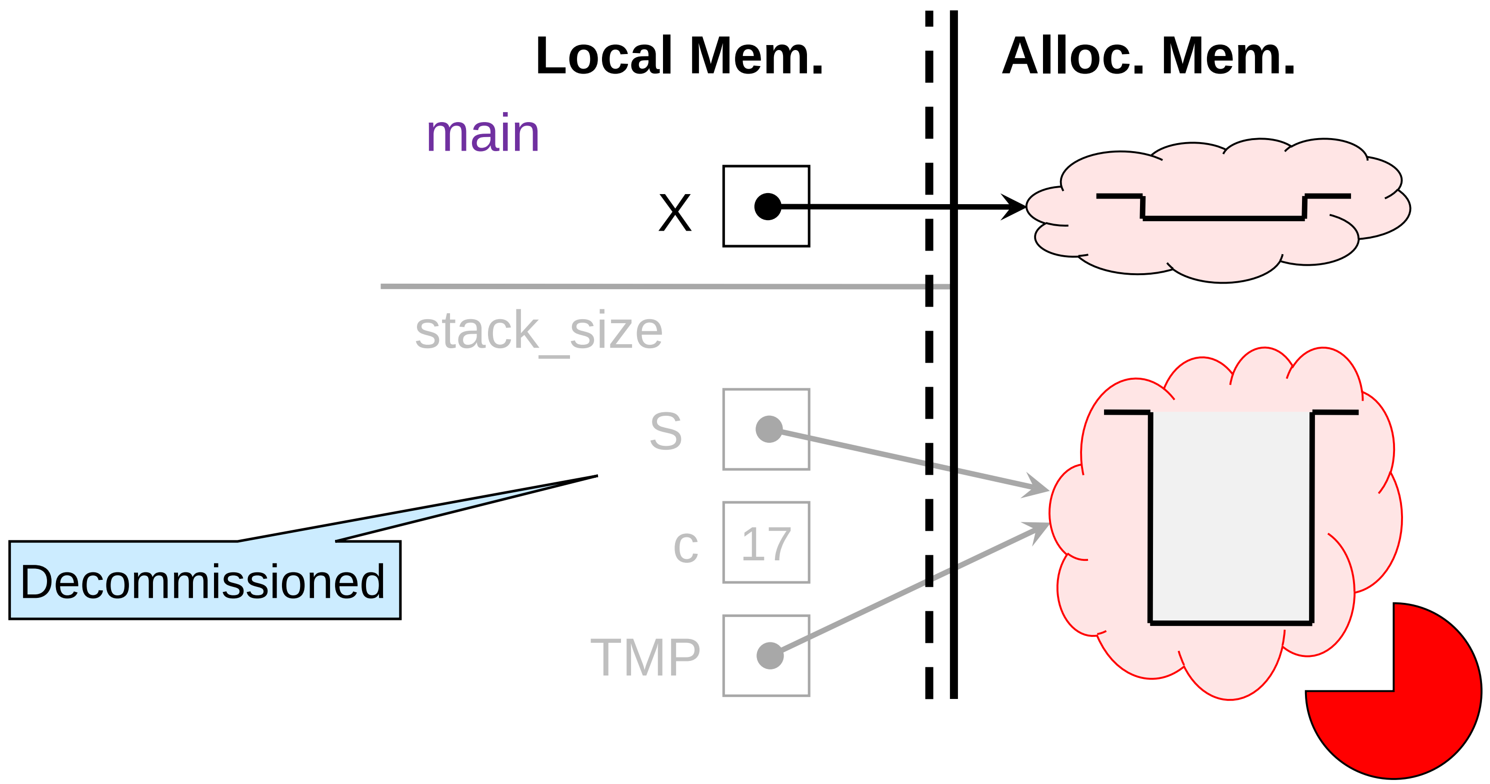
Aliasing!

The Size of a Stack

- On return, the caller stack is empty

```
int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  int c = 0;
  stack_t TMP = stack_new();
  while (!stack_empty(S)) {
    string x = pop(S);
    push(TMP, x);
    c++;
  }
  //@assert stack_empty(S);
  S = TMP;
  return c;
}                                v.2
```

```
int main() {
  …
  stack_t  X = stack_new();
  …
  … stack_size(X)
  …
  return 0;
}
```



- *Idea:*
  - We need to push contents of TMP back onto S
    - This will re-reverse it
    - restoring the original order of the elements in S

The Size of a Stack

Write a **client** function that returns the number of elements in a stack

- push elements back onto S

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;          @*/
  /*@ensures !stack_empty(S);     @*/ ;

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

Exercise: check that this code is safe

```
int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  int c = 0;
  stack_t TMP = stack_new();
  while (!stack_empty(S)) {
    string x = pop(S);
    push(TMP, x);
    c++;
  }
  //@assert stack_empty(S);
  while (!stack_empty(TMP)) {     // ADDED
     push(S, pop(TMP));           // ADDED
   }                              // ADDED
  //@assert stack_empty(TMP);     // ADDED
  return c;
}
```

v.3

- Does this do what we want?
  - This time yes!

✓

- What is the complexity?
  - We empty out the stack
    - twice
  - If S initially contains n elements, complexity is **O(n)**

The Size of a Stack

Write a **client** function that returns the number of elements in a stack

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

v.3

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;           @*/
  /*@ensures !stack_empty(S);      @*/ ;

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

- What is the complexity?
  - **O(n)**
- Can we do better?
  - not with *this* interface
  - but a good implementation could achieve O(1)
    - an interface that exports stack_size may provided it at cost O(1)

The Size of a Stack

Write a **client** function that returns the number of elements in a stack

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

v.3

Stack Interface

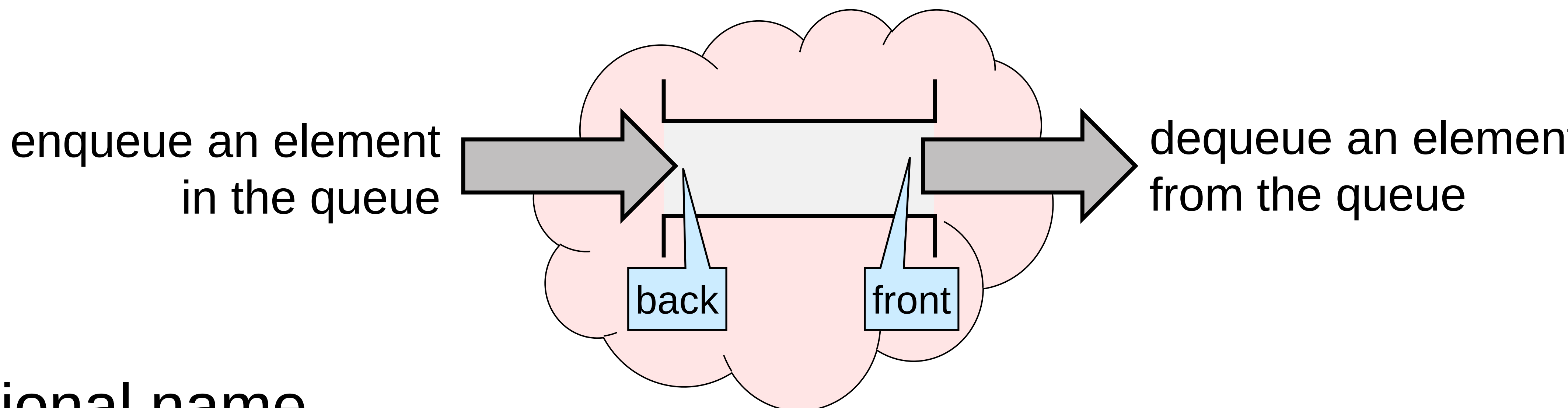
```
// typedef ______* stack_t;

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;            @*/
  /*@ensures !stack_empty(S);       @*/ ;

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

- Where are the loop invariants?
  - these loops have **no interesting invariants**!
  - this is because the implementation details are hidden behind the interface
    - as clients, we know too little
  - an implementation-side stack_size would have all the information to write meaningful loop invariants

Queues

Queues

- A worklist where we retrieve the element that has been there longest
  - **F**irst **I**n **F**irst **O**ut
  - Like a cafeteria line

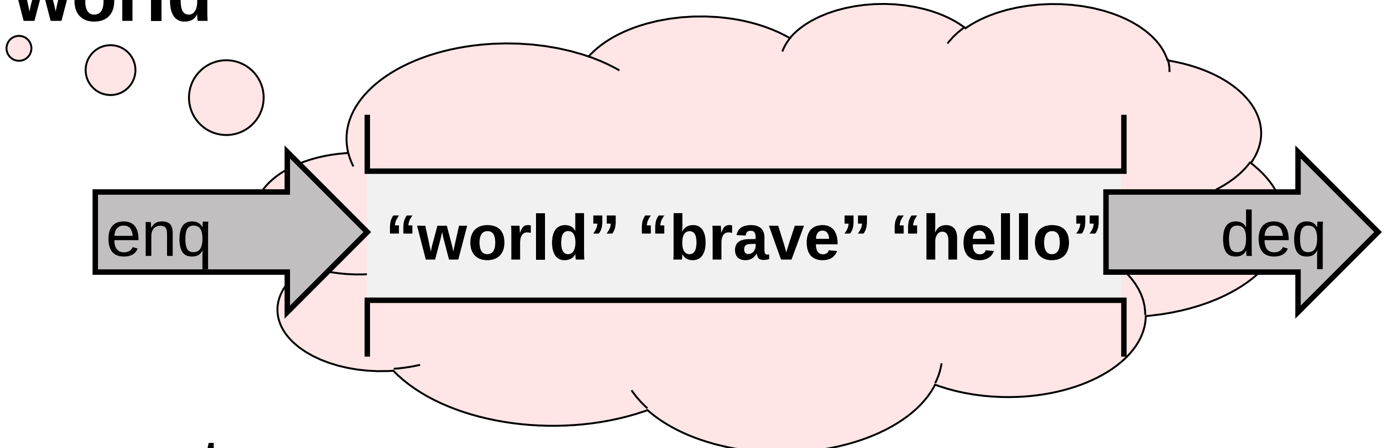


- Traditional name of operations
  - **enqueue** (= add) at the *back*
  - **dequeue** (= retrieve) from the *front*

Queues

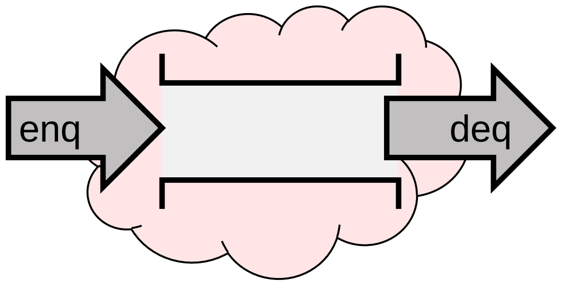
- A worklist where we dequeue the first element enqueued
  - **F**irst **I**n **F**irst **O**ut

- If we enqueue
  - **“hello”** then **“brave”** then **“world”**



- and then dequeue, we get
  - **“hello”**
- and then dequeue again, we get
  - **“brave”**
- and dequeue once more, we get
  - **“world”**
- at this point the queue is empty

The Queue Interface



Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)       // O(1)
  /*@requires S != NULL;              @*/
  /*@ensures !queue_empty(S);        @*/ ;

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

What

- This is again the worklist interface with the names changed
- This interface is also providing complexity bounds
  - all queue operations take constant time

Using only functions from the queue interface

Copying a Queue

Write a **client** function that returns a deep copy of a queue

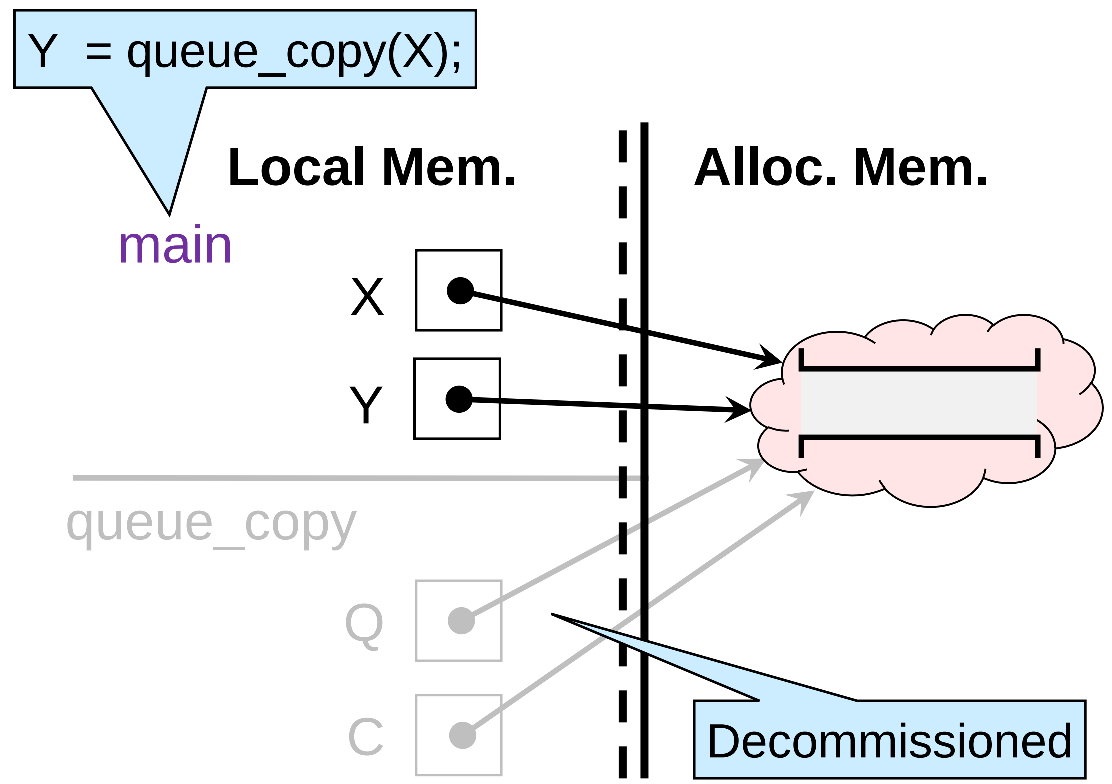
- a new queue with the same elements in the same order

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
{
  queue_t C = Q;
  return C;
}
```
v.1

- Does this do what we want?
  - it just returns an alias to Q! ✘
    - a *shallow copy*
  - **Idea:** we need to return a <u>new</u> queue

Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)   // O(1)
  /*@requires S != NULL;           @*/
  /*@ensures !queue_empty(S);      @*/ ;

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



Copying a Queue

Write a **client** function that returns a deep copy of a queue

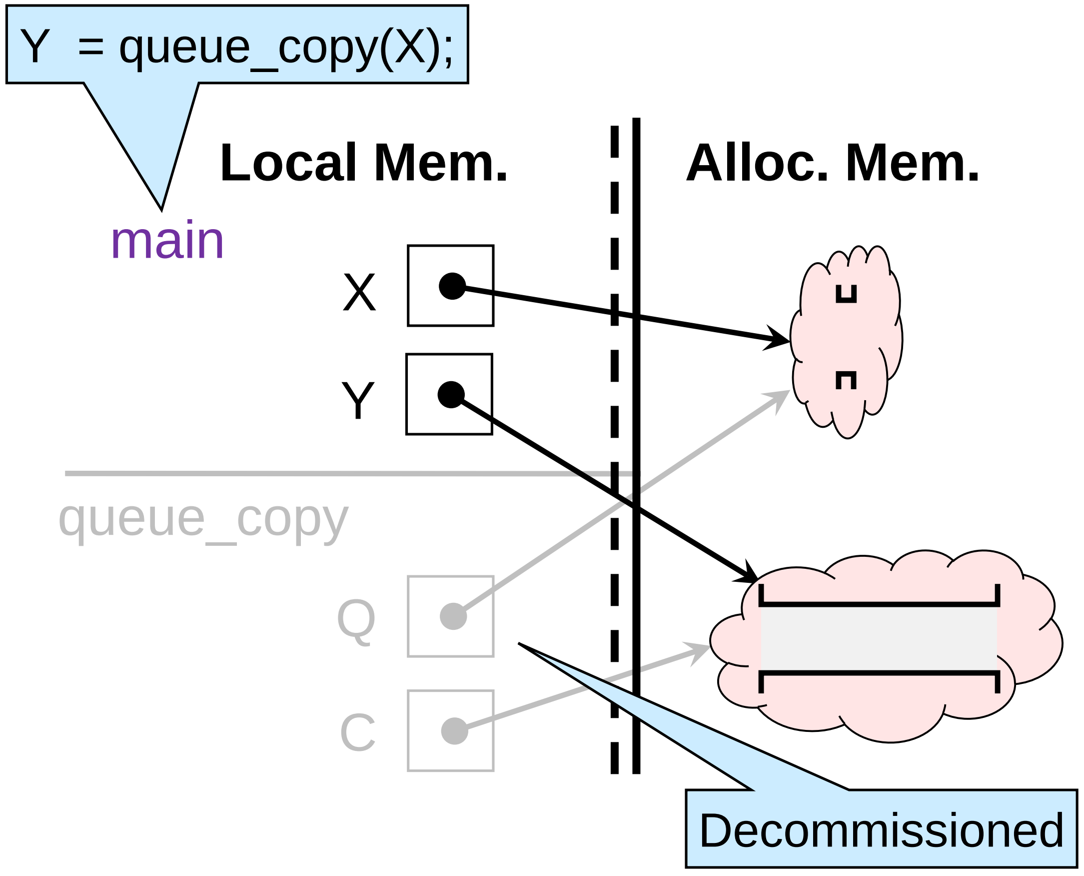
- return a new queue!

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
{
  queue_t C = new_queue();   // MODIFIED
  while (!queue_empty(Q)) {  // ADDED
    string x = deq(Q);       // ADDED
    enq(C, x);               // ADDED
  }
  return C;
}
```
v.2

- Does this do what we want?
  - it empties out Q ✘
  - **Idea:** put elements back onto Q!

Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)   // O(1)
  /*@requires S != NULL;          @*/
  /*@ensures !queue_empty(S);     @*/ ;

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



Copying a Queue

Write a **client** function that returns a deep copy of a queue

- put elements back into Q!

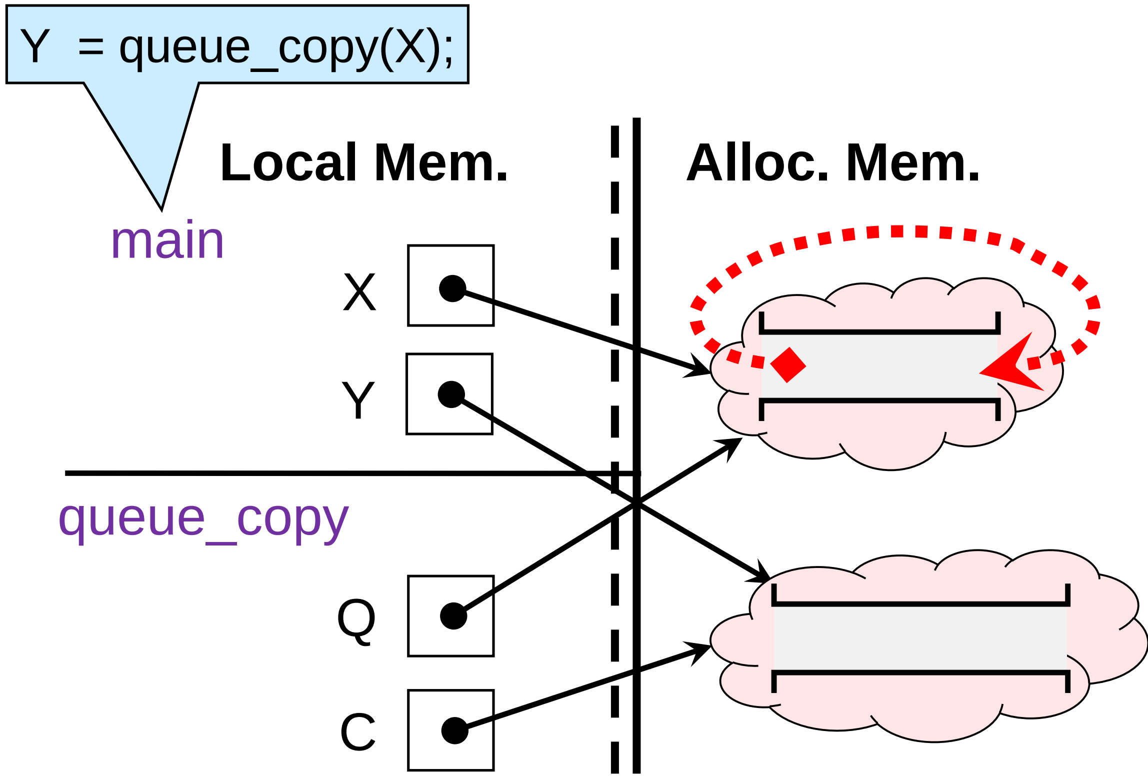
```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
{
  queue_t C = new_queue();
  while (!queue_empty(Q)) {
    string x = deq(Q);
    enq(C, x);
    enq(Q, x);                    // ADDED
  }
  return C;
}
```

v.3

- Does this do what we want?
  - it runs for ever! ✘
  - **Idea:** save elements in another queue

Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)    // O(1)
  /*@requires S != NULL;           @*/
  /*@ensures !queue_empty(S);      @*/ ;

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



Copying a Queue

Write a **client** function that returns a deep copy of a queue

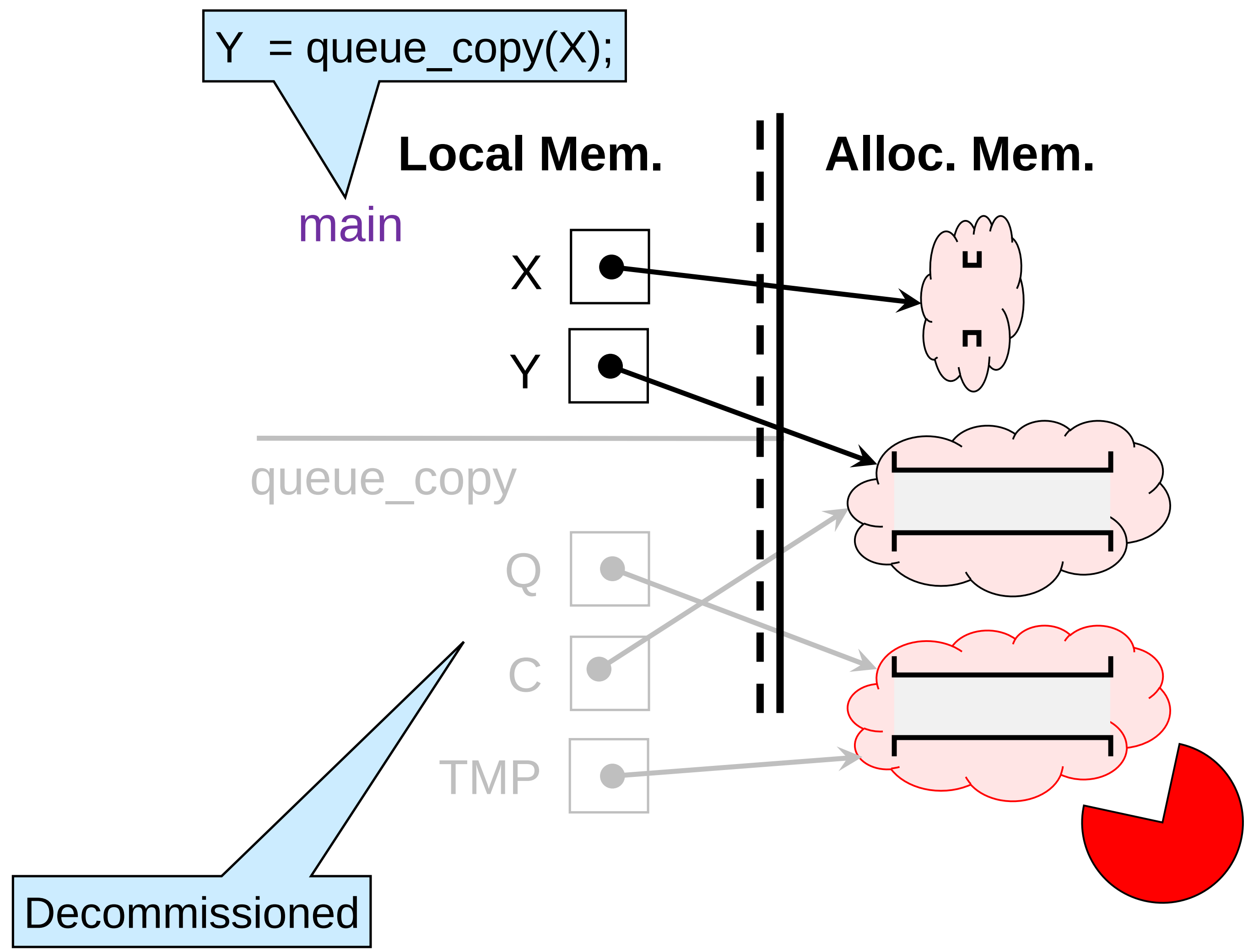
- save elements in another queue!

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
{
  queue_t C = new_queue();
  queue_t TMP = new_queue();  // ADDED
  while (!queue_empty(Q)) {
    string x = deq(Q);
    enq(C, x);
    enq(TMP, x);                // MODIFIED
  }
  //@assert queue_empty(Q);   // ADDED
  Q = TMP;                     // ADDED
  return C;
}                                       v.4
```

- Does this do what we want?
  - it empties out Q ✘

Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)   // O(1)
  /*@requires S != NULL;          @*/
  /*@ensures !queue_empty(S);     @*/ ;

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



Copying a Queue

Write a **client** function that returns a deep copy of a queue

- empty TMP back into Q

```
queue_t queue_copy(queue_t Q)
//@requires Q != NULL;
{
  queue_t C = new_queue();
  queue_t TMP = new_queue();
  while (!queue_empty(Q)) {
    string x = deq(Q);
    enq(C, x);
    enq(TMP, x);
  }
  //@assert queue_empty(Q);
  while (!queue_empty(TMP))      // ADDED
    enq(Q, deq(TMP));            // ADDED
  return C;
}
```

v.5

Queue Interface

```
// typedef ______* queue_t;

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

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

void enq(queue_t S, string x)    // O(1)
  /*@requires S != NULL;            @*/
  /*@ensures !queue_empty(S);       @*/ ;

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

- Does this do what we want?
  - This time yes!

✓

- What is the complexity?
  - We empty out the queue
    - twice
  - If Q initially contains n elements, complexity is **O(n)**

What have we done?

- We introduced two important types of worklists
  - Stacks
  - Queues
- We wrote **client code** based on their interface
- We dealt with
  - safety
  - aliasing
  - infinite loops
- We determined the complexity of client code based on the known cost of library functions