Stacks

Stacks

Stacks are simple—but very useful—data structures

- they are ordered containers where entries can only be inserted and removed from one end (called the top of the stack)
- to access the item at the bottom of a stack, you must first remove all items before it
- a stack is a last-in, first-out data structure; entries are removed from the stack in the reverse order of their insertion

Picturing a stack is easy:

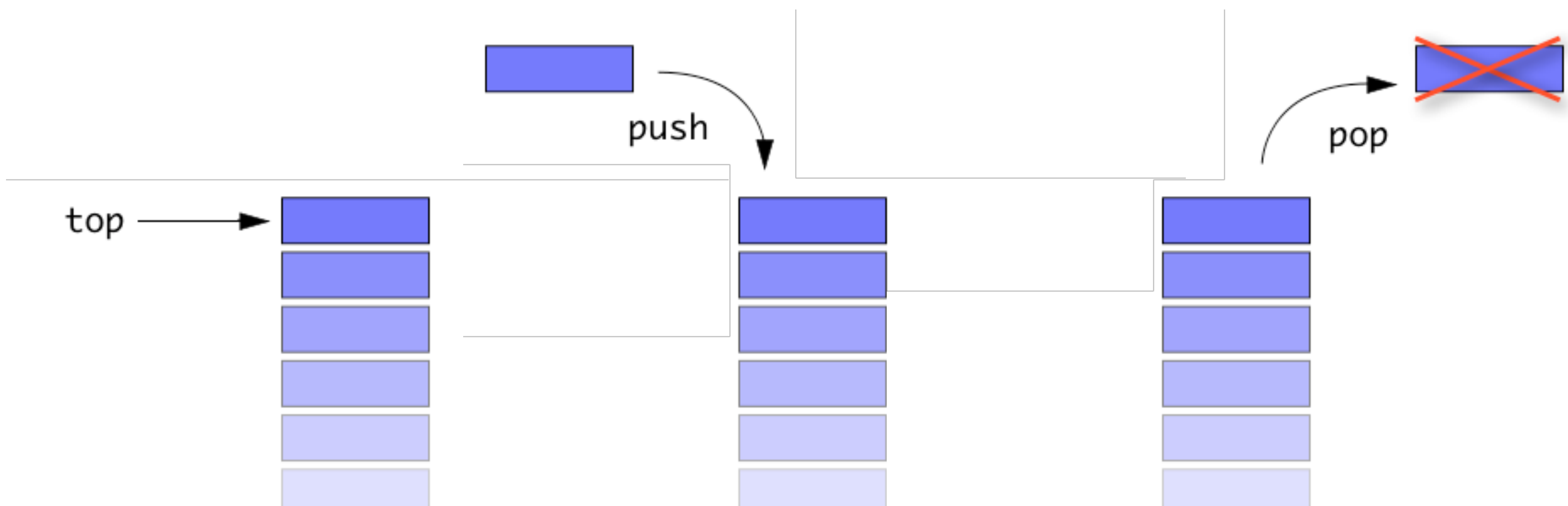
`top of the stack`

Stacks

Stacks support three basic operations (all at the top of the stack):

- `top`: access the item at the top of the stack
- `push`: add an item to the top of the stack
- `pop`: remove the item at the top of the stack

Visualizing these operations:



The STL stack

The STL has a stack class you can use

- like other STL containers, it is a template class and can contain items of any type
- we'll later see several different implementations of stack classes

Member functions

(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Add element (public member function)
pop	Remove element (public member function)

The STL stack

What does the following code output?

```
string text = "Data structures";
stack<char> letters;

for (size_t i = 0; i < text.length(); i++) {
    letters.push( text[i] );
}

while (!letters.empty()) {
    cout << letters.top();
    letters.pop();
}
```

Practical Applications of Stacks

Despite their simplicity, stacks have many useful applications

Many compilers use stacks to analyze the syntax of programs:

- stacks are useful to make sure parentheses and braces are matched as they should be (as we'll see)
- they're also useful for tracking operands, operators, symbols...

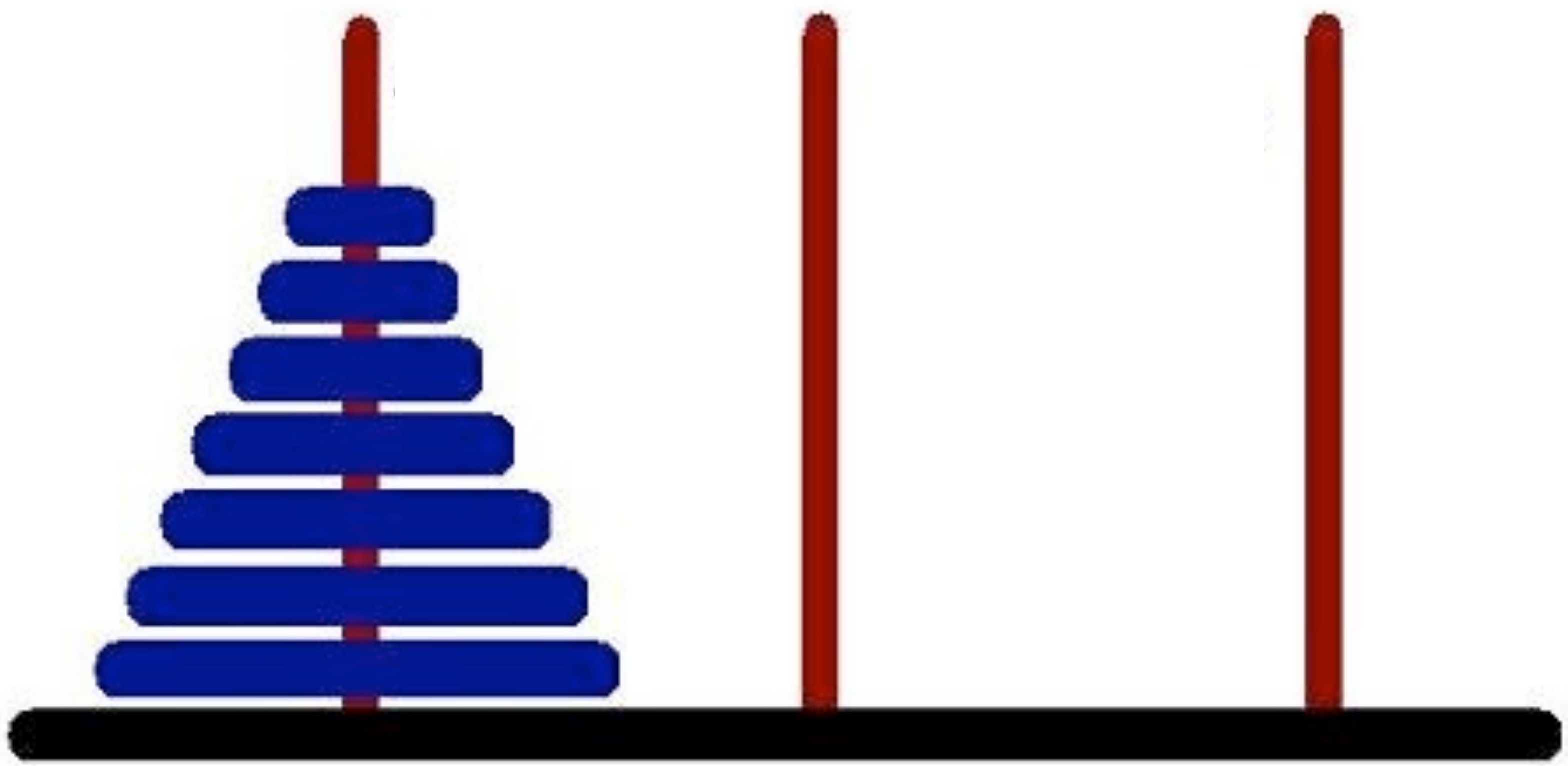
Stacks are well suited to traversing / searching branching structures

- this includes trees, mazes, etc...

Practical Applications of Stacks

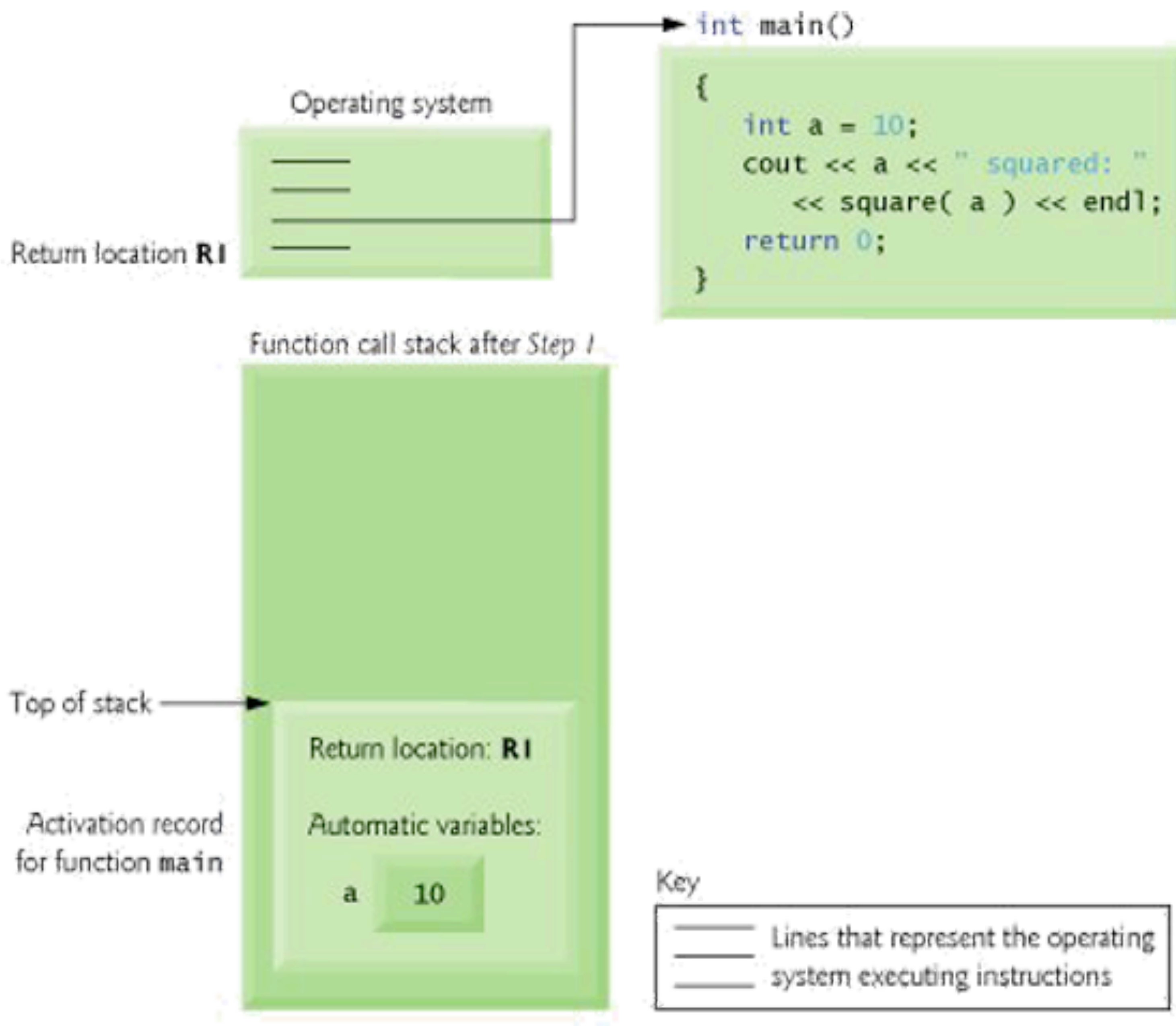
Despite their simplicity, stacks have many useful applications

How could you use stacks to solve the Tower of Hanoi game?



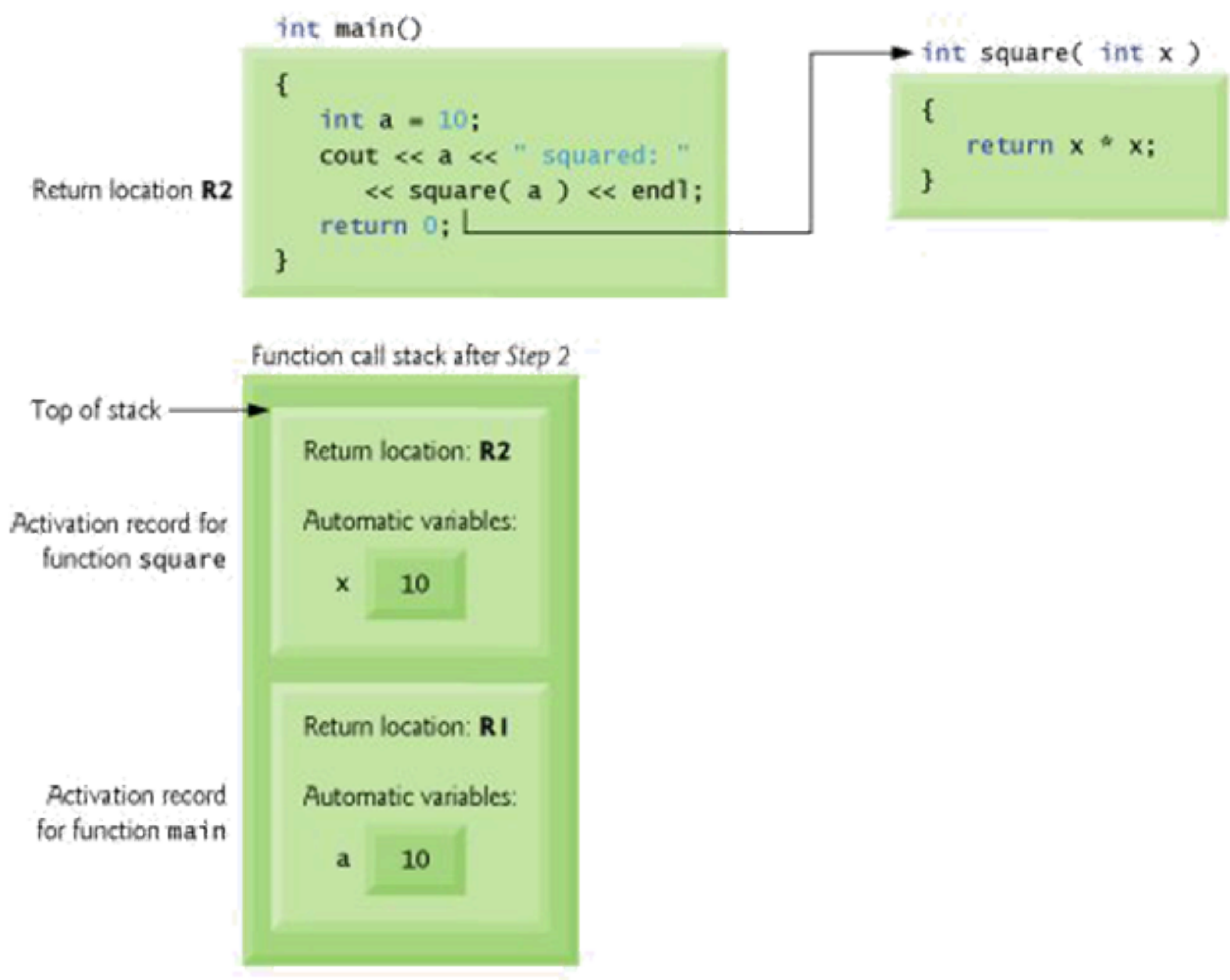
Practical Applications of Stacks

Programs use a stack to track function calls (the call stack):



Practical Applications of Stacks

Programs use a stack to track function calls (the call stack):



Stack Implementation: Array Version

Specification

Stack: Array Version

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Item>

// Item is the type of items in the stack, and must be
// a built-in type, or a class that provides:
//  - instantiation via a default constructor
//  - instantiation via a copy constructor
//  - assignment operator   (x = y)
```

Stack: Array Version

Template parameters, typedefs, and member constants:

```
// alias for the template parameter
typedef Item value_type;
```

Stack: Array Version

Template parameters, typedefs, and member constants:

```
// data type of variables that track the stack's size
typedef std::size_t size_type;
```

Stack: Array Version

Template parameters, typedefs, and member constants:

```
// the maximum capacity for any stack
static const size_type CAPACITY = 30;

// stack<Item>::CAPACITY is the maximum capacity of
// any stack; once CAPACITY is reached, further pushes
// are forbidden
```

Stack: Array Version

Constructors:

```
// creates an empty stack
stack();

// postcondition:
//   the stack has been initialized as an empty stack
```

Stack: Array Version

Modification member functions:

```
// pushes @entry onto the top of the stack
void push(const Item& entry);

// precondition:
//   size() < CAPACITY
// postcondition:
//   A new copy of entry has been pushed onto the stack
```

Stack: Array Version

Modification member functions:

```
// pops the top item off the stack
void pop();


// precondition:
//   size() > 0
// postcondition:
//   The top item of the stack has been removed
```

Stack: Array Version

Constant member functions:

```
// returns the top item of the stack
Item top() const;

// precondition:
//   size() > 0
// postcondition:
//   The return value is the top item of the stack, but
//   the stack is unchanged. This differs slightly from
//   the STL stack, where the top function returns a
//   reference to the item on the top of the stack
```

Stack: Array Version

Constant member functions:

```
// returns the total number of items in the stack
size_type size() const;

// postcondition:
//   The return value is the total number of items in
//   the stack
```

Stack: Array Version

Constant member functions:

```
// returns true if the stack is empty, false otherwise
bool empty() const;

// postcondition:
//   The return value is true if the stack is empty, and
//   false otherwise
```

Stack: Array Version

Value semantics:

```
// stack<Item> objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Stack Implementation: Array Version

Implementation

Stack: Array Version

Starting the header file:

```
// stack.h header file
// specification documentation
#pragma once
#include <cstdlib>

namespace CS262 {
    template <typename Item>
    class stack { };
}

#include "stack.cpp"
```

Stack: Array Version

Starting the implementation file:

```
// stack.cpp implementation file
// INVARIANT (coming soon)
#include <cassert>
using namespace std;

namespace CS262 {
    // member implementations
}
```

Stack: Array Version

Type definitions and member constants:

```
template <typename Item>
class stack {
    public:
        typedef std::size_t size_type;
        typedef Item value_type;
        static const size_type CAPACITY = 30;

    private:
        Item data[CAPACITY];
        size_type used;
};
```

Stack: Array Version

Document invariant in implementation file:

```
// stack.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for stack<Item> class:
// 1. The number of items in the stack is stored in the
//    member variable used
// 2. The items in the stack are stored in a partially
//    filled array called data, with the bottom of the
//    stack at data[0], the next entry at data[1], and
//    so on to the top of the stack at data[used - 1].
```

Stack: Array Version

The constructor (inline implementation):

```
// creates an empty stack
stack() : used(0) { }
```

Stack: Array Version

The size method (inline implementation):

```
// returns the total number of items in the stack
size_type size() const { return used; }
```

Stack: Array Version

The empty method (inline implementation):

```
// returns true if the stack is empty, false otherwise
bool empty() const { return (used == 0); }
```

Stack: Array Version

The push function prototype:

```
// pushes @entry onto the top of the stack
void push(const Item& entry);
```

Implementation:

```
template <typename Item>
void stack<Item>::push(const Item& entry) {
    assert(size() < CAPACITY);
    data[used] = entry;
    used++;
}
```

Stack: Array Version

The pop function prototype:

```
// pops the top item off the stack
void pop();
```

Implementation:

```
template <typename Item>
void stack<Item>::pop() {
    assert(!empty());
    used--;
}
```

Stack: Array Version

The top function prototype:

```
// returns the top item of the stack
Item top() const;
```

Implementation:

```
template <typename Item>
Item stack<Item>::top() const {
    assert(!empty());
    return data[used - 1];
}
```

Stack: Array Version

The array version doesn't use dynamic memory...

- this means that the automatic versions of the copy constructor, assignment operator, and destructor are just fine! Hooray!

Stack Implementation: Linked-List Version

Specification

Stack: Linked-List Version

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Item>

// Item is the type of items in the stack, and must be
// a built-in type, or a class that provides:
//  - instantiation via a default constructor
//  - instantiation via a copy constructor
//  - assignment operator   (x = y)
```

Stack: Linked-List Version

Template parameters, typedefs, and member constants:

```
// alias for the template parameter
typedef Item value_type;
```

Stack: Linked-List Version

Template parameters, typedefs, and member constants:

```
// data type of variables that track the stack's size
typedef std::size_t size_type;
```

Stack: Linked-List Version

Constructors:

```
// creates an empty stack
stack();


// postcondition:
//   the stack has been initialized as an empty stack
```

Stack: Linked-List Version

Modification member functions:

```
// pushes @entry onto the top of the stack
void push(const Item& entry);

// postcondition:
//   A new copy of entry has been pushed onto the stack
```

Stack: Linked-List Version

Modification member functions:

```
// pops the top item off the stack
void pop();


// precondition:
//   size() > 0
// postcondition:
//   The top item of the stack has been removed
```

Stack: Linked-List Version

Constant member functions:

```
// returns the top item of the stack
Item top() const;

// precondition:
//   size() > 0
// postcondition:
//   The return value is the top item of the stack, but
//   the stack is unchanged. This differs slightly from
//   the STL stack, where the top function returns a
//   reference to the item on the top of the stack
```

Stack: Linked-List Version

Constant member functions:

```
// returns the total number of items in the stack
size_type size() const;

// postcondition:
//    The return value is the total number of items in
//    the stack
```

Stack: Linked-List Version

Constant member functions:

```
// returns true if the stack is empty, false otherwise
bool empty() const;

// postcondition:
//   The return value is true if the stack is empty, and
//   false otherwise
```

Stack: Linked-List Version

Value semantics:

```
// stack<Item> objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Stack: Linked-List Version

Dynamic memory usage:

```
// If there is insufficient dynamic memory, then the
// following functions throw bad_alloc:
//   copy constructor
//   push
//   operator =
```

Stack Implementation: Linked-List Version

Implementation

Stack: Linked-List Version

Starting the header file:

```
// stack.h header file
// specification documentation
#pragma once
#include <cstdlib>
#include "Node.h"

namespace CS262 {
    template <typename Item>
    class stack { };
}
#include "stack.cpp"
```

Stack: Linked-List Version

Starting the implementation file:

```
// stack.cpp implementation file
// INVARIANT (coming soon)
#include <cassert>
using namespace std;

namespace CS262 {
    // member implementations
}
```

Stack: Linked-List Version

Type definitions and member constants:

```
template <typename Item>
class stack {
    public:
        typedef std::size_t size_type;
        typedef Item value_type;

    private:
        Node<Item>* top_ptr;
        size_type used;
};
```

Stack: Linked-List Version

Document invariant in implementation file:

```
// stack.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for stack<Item> class:
// 1. The number of items in the stack is stored in the
//    member variable used.
// 2. The items in the stack are stored in a linked
//    list, with the top of the stack stored at the head
//    node, down to the bottom at the final node.
// 3. The member variable top_ptr is the head pointer of
//    the linked list.
```

Stack: Linked-List Version

The constructor (inline implementation):

```
// creates an empty stack

stack() : used(0), top_ptr(NULL) { }
```

Stack: Linked-List Version

The size method (inline implementation):

```
// returns the total number of items in the stack
size_type size() const { return used; }
```

Stack: Linked-List Version

The empty method (inline implementation):

```
// returns true if the stack is empty, false otherwise
bool empty() const { return (top_ptr == NULL); }
```

Stack: Linked-List Version

The push function prototype:

```
// pushes @entry onto the top of the stack
void push(const Item& entry);
```

Implementation:

```
template <typename Item>
void stack<Item>::push(const Item& entry) {
    top_ptr = new Node<Item>(entry, top_ptr);
    used++;
}
```

Stack: Linked-List Version

The pop function prototype:

```
// pops the top item off the stack
void pop();
```

Implementation:

```
template <typename Item>
void stack<Item>::pop() {
    assert(!empty());
    list_head_remove(top_ptr);
    used--;
}
```

Stack: Linked-List Version

The top function prototype:

```
// returns the top item of the stack
Item top() const;
```

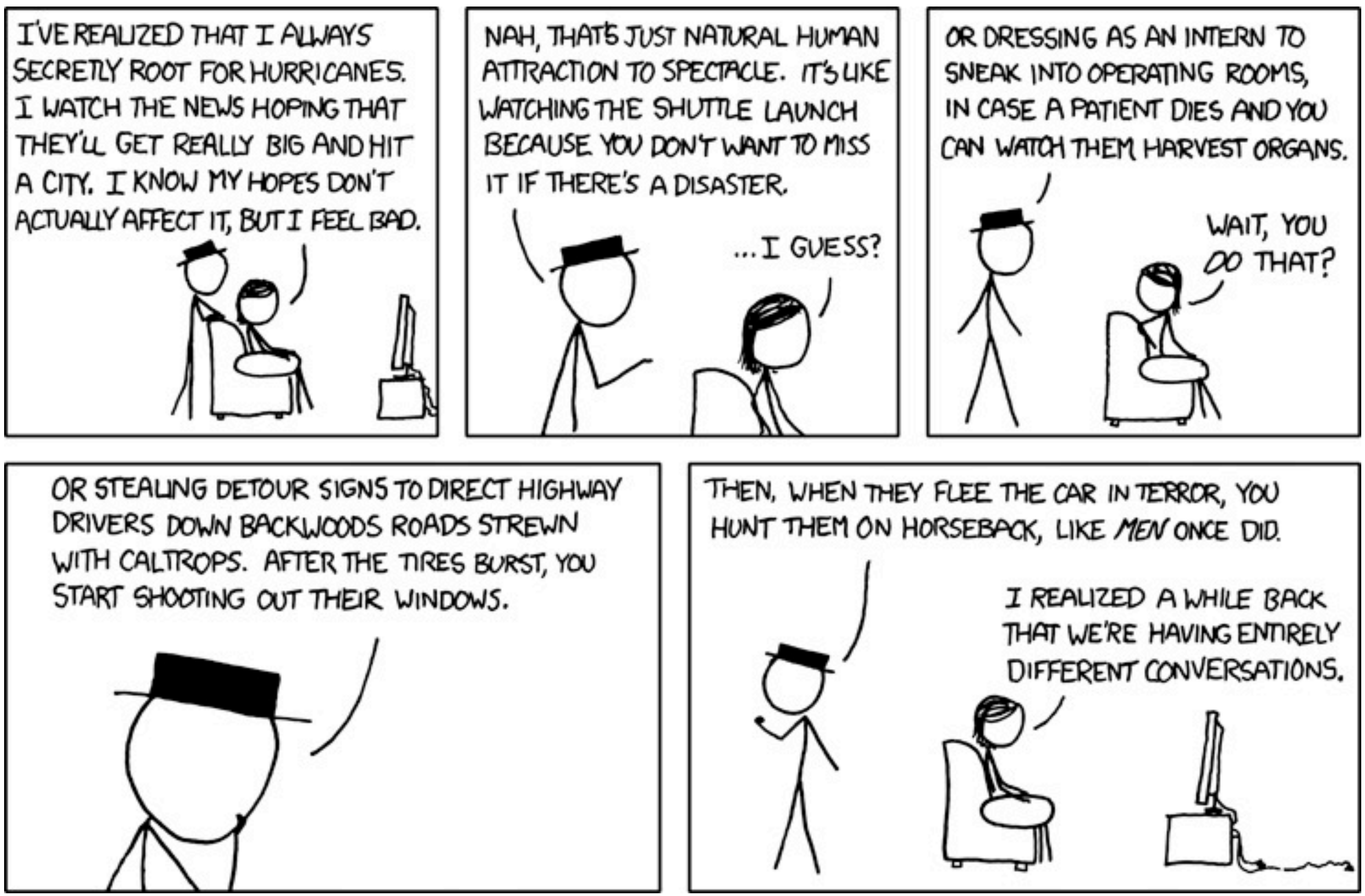
Implementation:

```
template <typename Item>
Item stack<Item>::top() const {
    assert(!empty());
    return top_ptr->data();
}
```

Stack: Linked-List Version

The linked-list version <u>does</u> use dynamic memory...

- we need to implement our own copy constructor, assignment operator, and destructor



Stack: Linked-List Version

The copy constructor prototype:

```
// creates a new stack as a copy of @source
stack(const stack& source);
```

The copy constructor must:

- copy all nodes from the source stack to the new one
- update the value of `used`

Stack: Linked-List Version

The copy constructor implementation:

```
template <typename Item>
stack<Item>::stack(const stack<Item>& source) {
    top_ptr = NULL;
    used = source.used;
    Node<Item> *n, *tail = NULL;

    for (n = source.top_ptr; n != NULL; n = n->link()) {
        tail = new Node<Item>(n->data(), NULL, tail);
        if (top_ptr == NULL) top_ptr = tail;
    }
}
```

Stack: Linked-List Version

The assignment operator prototype:

```
// assigns this stack as a copy of @src
stack& operator =(const stack& src);
```

The assignment operator must:

- check for self-assignment
- clear the existing stack
- copy all nodes from the source stack
- update the value of `used`
- ideally, it should also return the calling object by reference (for assignment chaining)

Stack: Linked-List Version

The assignment operator implementation:

```
template <typename Item>
stack<Item>& stack<Item>::operator =(const stack& src) {
    if (this == &src) return *this;

    // clear existing list

    // copy data from src list as in copy constructor

    return *this;
}
```

Stack: Linked-List Version

The destructor prototype:

```
// frees the memory used by this stack
~stack();
```

The destructor must:

- yeah, that... =)

Stack: Linked-List Version

The destructor implementation:

```
template <typename Item>
stack<Item>::~stack() {
    while (top_ptr != NULL) {
        Node<Item>* remove_ptr = top_ptr;
        top_ptr = top_ptr->link();
        delete remove_ptr;
    }
}
```