Stacks and Queues

and a quick note on exceptions

Exception Handling

- mechanism to handle exceptional (unusual behavior)
  - –I/O problem
  - –subscript out of range
  - –violation of a precondition
- Allow clean design of reliable code (avoids cluttering the “usual” code with lots of special cases
- in C++ exceptions thrown in a “try” block are caught in a “catch” statement or propagated to block in which this is nested, or propagated to the caller

Exception Handling

- Try blocks
  - enclose a throw expression/call to a function that throws an exception inside a try block
- Catch blocks
  - follows a try block or call to a function that has a try block
- Throw expressions
  - flag an unusual situation
  - is of type void
  - can pass information “back”

http://www.cplusplus.com/doc/tutorial/exceptions/

```
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }

  try {
    // code here
  }
  catch (int param) { cout << "int exception"; }
  catch (char param) { cout << "char exception"; }
  catch (...) { cout << "default exception"; }

  return 0;
}
```

```
void g()
{
    throw std::exception();
}

void f()
{
    std::string str = "Hello"; // This string is newly allocated
    g();
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }
}
```

https://en.wikibooks.org/wiki/C++_Programming/Exception_Handling

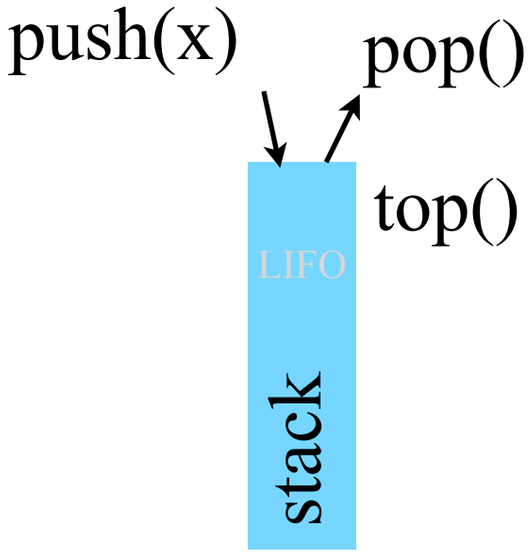
```
try
{   ...
     v.pop_back();
     ...  // assuming pop_back was OK
}
catch (const UnderFlowException & e)
{
     cout << e.what() << endl;
     ....
}
```

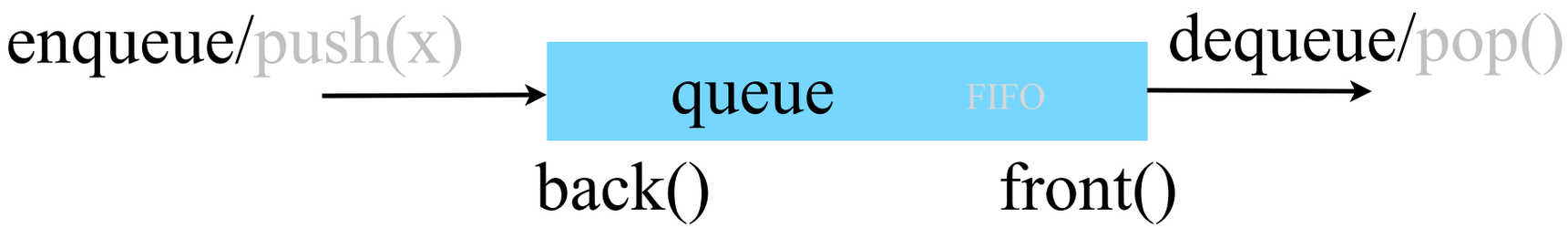
Abstract Data Types

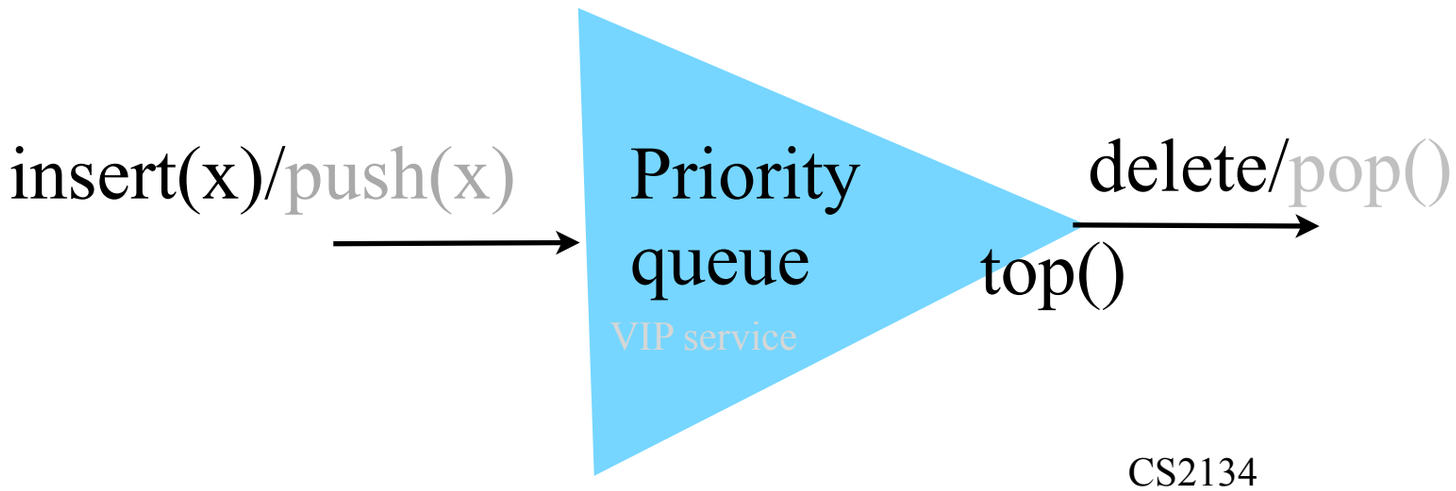
– Abstract Data Types
  - Abstract description of the operations provided and the relationships among them
  - Different implementations are possible for the same ADT
    - Separation of concerns between data type implementation and use
    - Were designed around common algorithmic constructs, rather than physical design
– Classes in Object Oriented languages group data (member variables) with operations to manipulate the data (member functions)
– OO languages developed to support ADTs

ADT's

stack, queue, priority queue



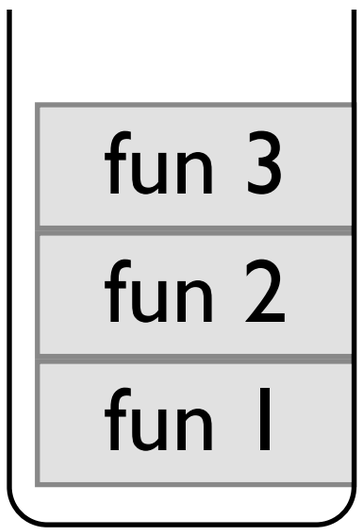




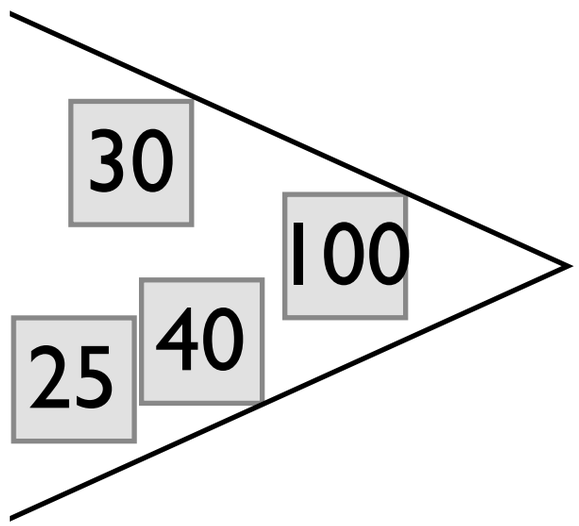
The ADT stack will be used when we design a simple calculator, do a simple balanced paren checking.

The ADT's queue, priority queue, and stack will be used as a subroutine in a graph algorithm

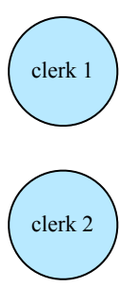
Function Calls



Printer Jobs



customersInLine



How would we implement these?

computer 1

computer 2

printer queue

printer

Sue's	Sue's	Sam's

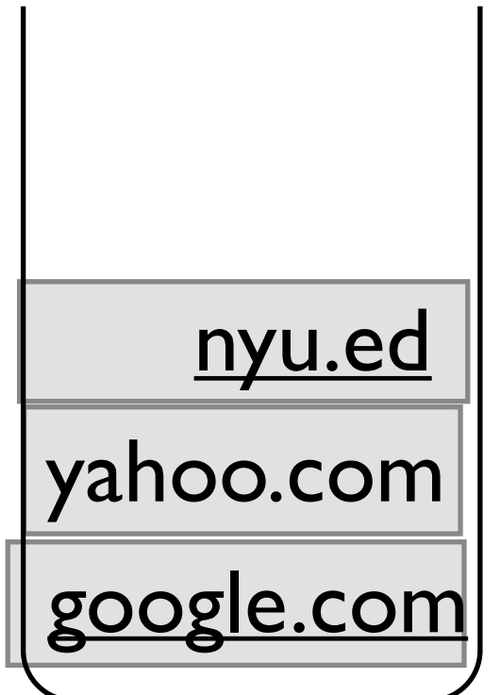
stored video file

video buffering

p3	p2	p1

video player

Back Button for Web Pages



How would we implement these?

STL Container Adapters

Container Adapter

"A container adaptor provides a different (typically restricted) interface to a container. Container adaptors are intended to be used only through their specialized interfaces. In particular, the STL container adaptors do not offer direct access to their underlying container. They do not offer iterators or subscripting."

Stroustrup, Bjarne

Simpler - the underlying implementation is not apparent to the user - less details for the user to understand

Flexible - the implementation can vary for different compilers - the user works through the interface and does not need to be aware of differences

Safer - the user can be prevented from some errors

Stacks

Stacks

- Sequence of data items with access only at one end, the top
  - –think of stack of trays in a cafeteria
- Stack operations:

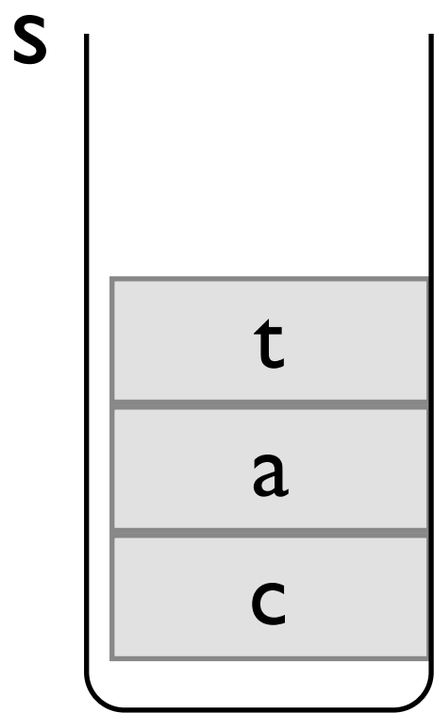
```
const Object& top( ) const;
void pop( );
void push( const Object & x );
bool empty( ) const;
```

Stack
Last in first out (LIFO)

- elements are removed (popped) in the reverse order that they're put in
- Useful for
  - reversing a sequence
  - memory management for function calls (stack of activation records)
  - checking that parentheses are balanced
  - evaluating expressions

Reversing a sequence

```
stack<char> s;
char x;
cin >> noskipws;  //read whitespace
cin >> x;
while (x != ' ')
{
    s.push(x);
    cin>> x;
}

// Now all the characters in current
// word are on the stack.
// Next, pop them off and output them
while (!s.empty( ))
{
    cout << s.top( );
    s.pop( );
}
cout << endl;
```

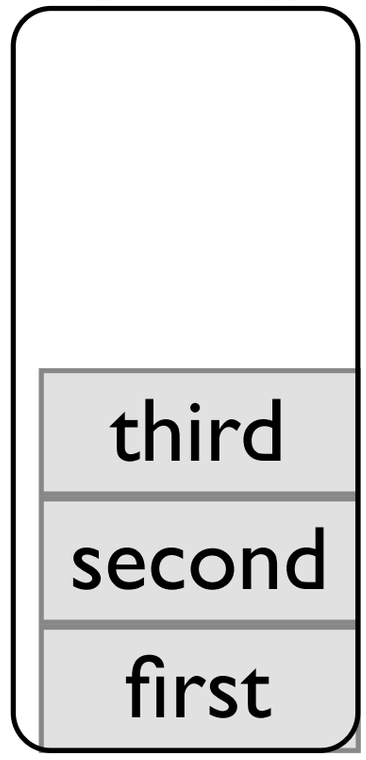


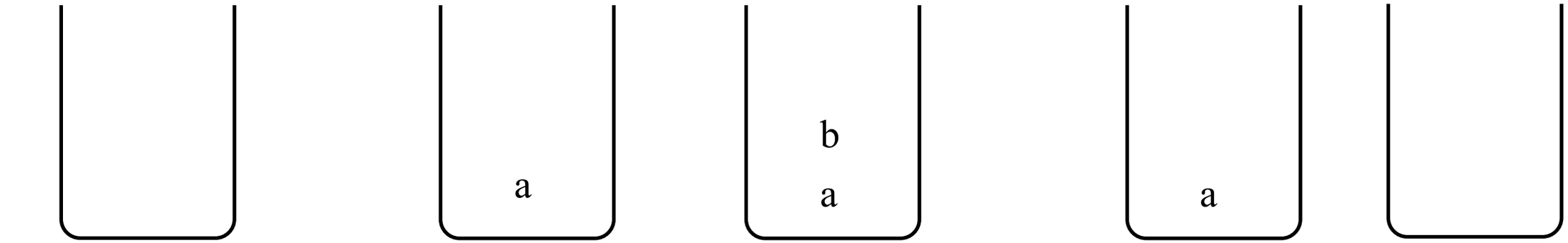
There are many possible ways to implement the ADT stack

STL stack class

```
#include <stack>
using namespace std;
int main ()
{
  stack<string> myStackOfNames;
  myStackOfNames.push("first");
  myStackOfNames.push("second");
  myStackOfNames.push("third");
  cout<< "stack size is "<<   myStackOfNames.size()<<endl;
  cout<< "Top of the stack:"<<  myStackOfNames.top()<<endl;
  cout<< "stack size is "<<   myStackOfNames.size()<<endl;
  myStackOfNames.pop();
  myStackOfNames.pop();
```

myStackOfNames





How to Implement a Stack?

Requirement: Constant Time Operations

Thinking about ADT stack

push(a)

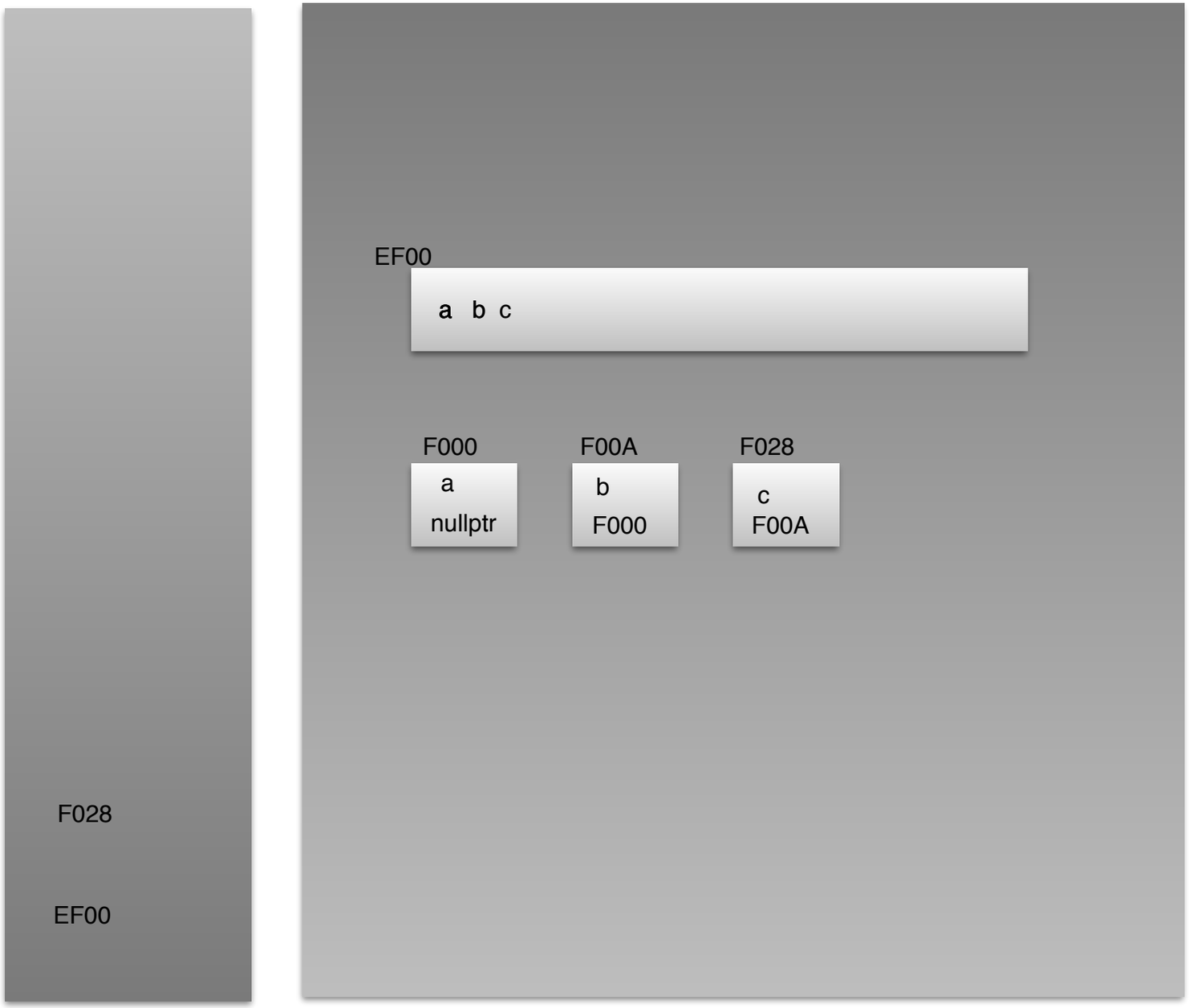
push(b)

push(c)

pop()

pop()

pop()

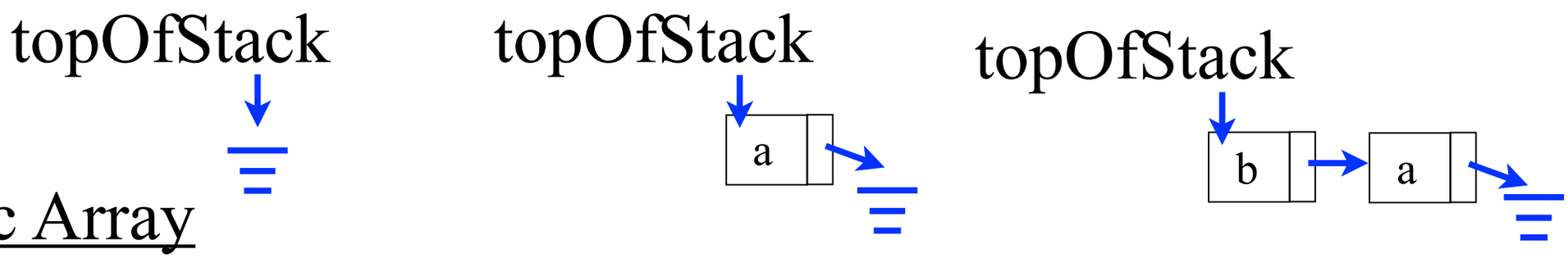


ADT

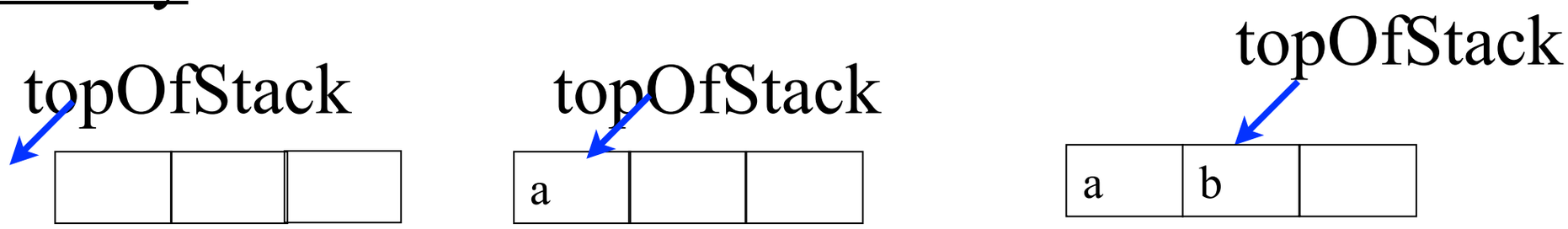
empty stack push(‘a’) push(‘b’) pop( ) pop( )

Conceptual Representation

Singly Linked List

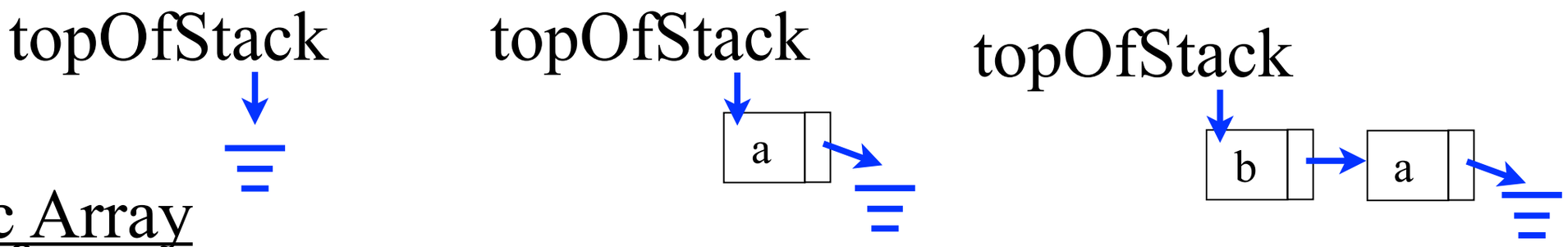


Static Array & Dynamic Array

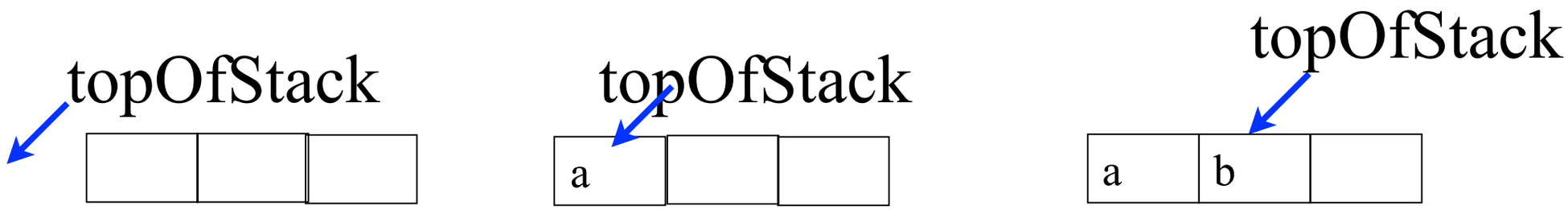


Conceptual Representation

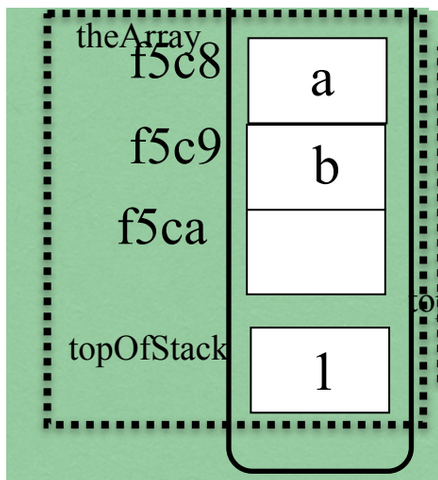
Singly Linked List



Static Array
&
Dynamic Array

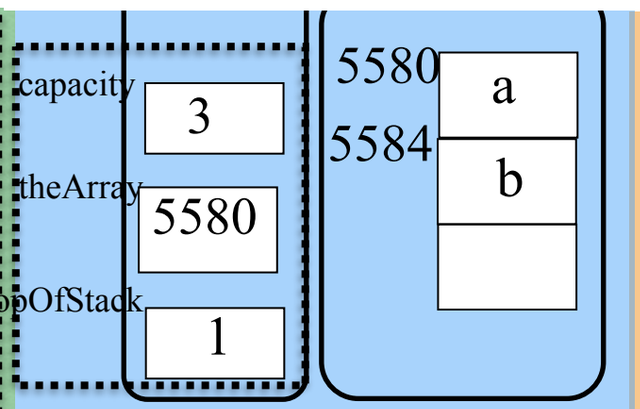


Memory Level



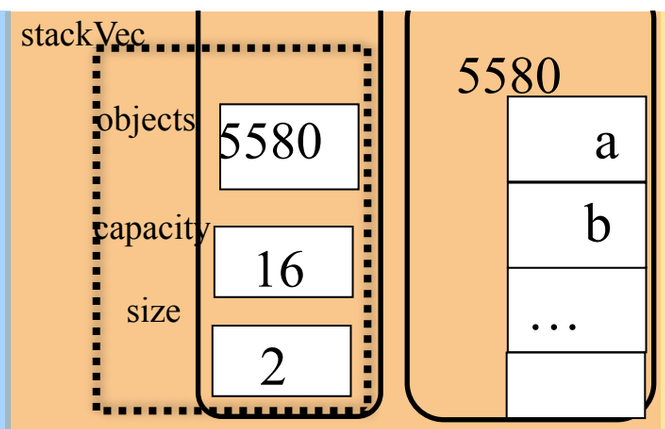
Static Array

Object theArray[MAX];
int topOfStack;

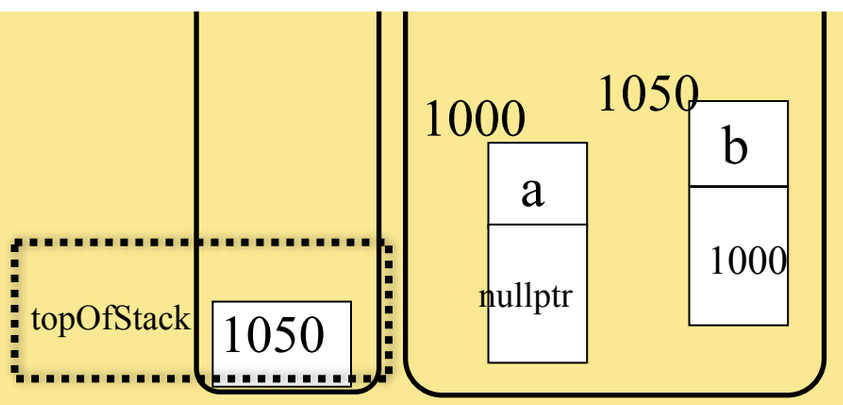


Dynamic Array

Object * theArray;
int topOfStack;
int capacity;



Dynamic Array using aVector



Singly Linked List

struct ListNode
{ Object element;
ListNode* next; }

ListNode *topOfStack;

Stack Interface:

```
template< class Object >
class Stack
{
private:
    ....
public:
    Stack( );

    bool empty( ) const;

    const Object& top( )const;
    void push (const Object& x);
    void push (Object && x);
    void pop( );
};
```

NOT a complete class!

Stack Implementation 1: Static Array

- data members:

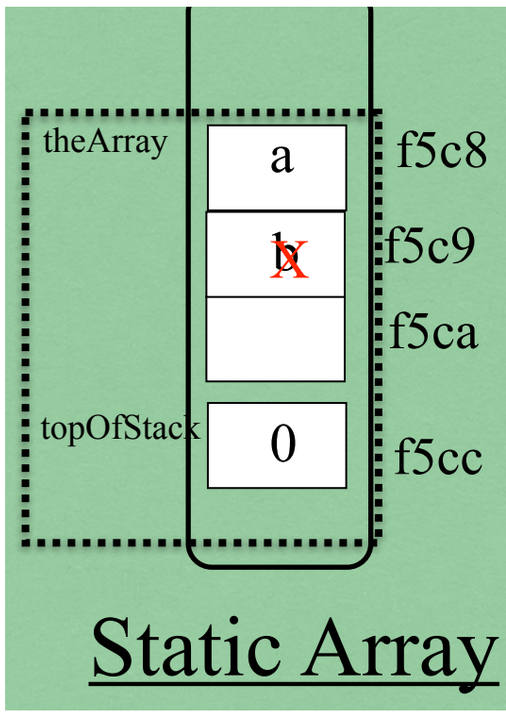
  Object theArray[MAX];

  int topOfStack;

- representation invariant:
  - stack elements stored in slots 0 to topOfStack, from bottom (at 0) to top.
- operations:
  - push(x): theArray[++topOfStack] = x;
    - must avoid stack overflow
  - pop( ): topOfStack--
    - must avoid stack underflow
  - top( ): return theArray[topOfStack];
    - must avoid stack underflow
- each operation is O(1)

Stack<char> s;

s.push('a');
s.push('b');
s.pop( );
s.top( );



Array Based Stack


```
template< class Object >
class Stack
{
    enum { MAX = 100 };
private:
    Object theArray[MAX];
     int topOfStack;
public:
    Stack( ) :topOfStack(-1){ }

    bool empty( ) const  {return topOfStack == -1;}
    int size( ) const  {return topOfStack + 1;}

    const Object & top( )const
        {if (empty()) throw underflow_error("Top of empty stack
        return theArray[topOfStack]; }
    void push (const Object& x)     //also add push(Object && x)
        {if ( MAX == size()) throw overflow_error("overflow");
         theArray[++topOfStack]=x;}
    void pop( )
        {if (empty()) throw underflow_error("empty stack");
         --topOfStack;}
};
```

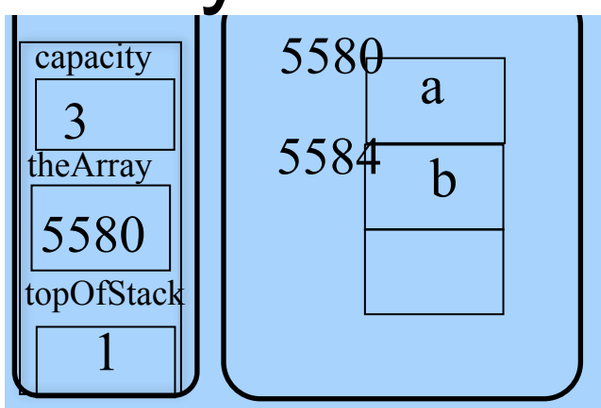
```
Stack<char> s;
s.push('a');
s.top( );
s.push('b');
s.pop( );
```

Stack Implementation 2: Dynamic Array

- data members:

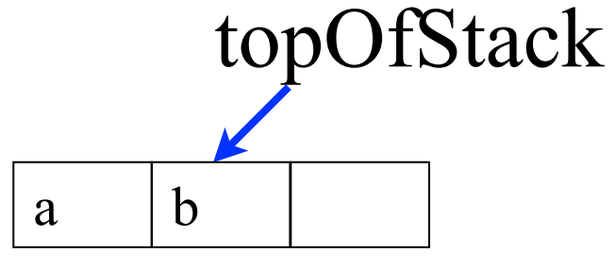
```
Object * theArray;
int    topOfStack;
int    capacity;
```

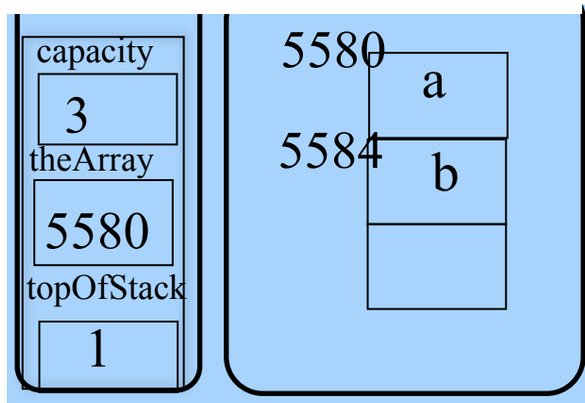
- operations same as static array version, except that array is doubled when necessary to prevent overflow.
- each operation is O(1), except push() when array doubling is done
  - this is rare so “amortized time” is O(1).



Dynamic Array

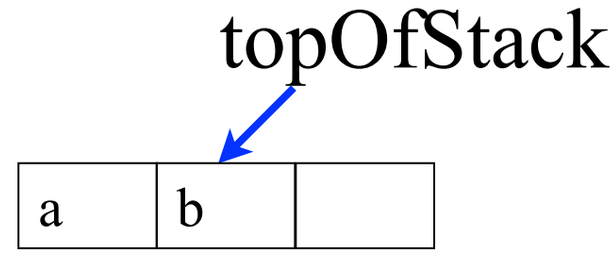
```
template< class Object >
class Stack
{
 enum { DEF_CAPACITY = 3 };
 private:
    Object * theArray;
    int topOfStack;
    int capacity;
    Object & increaseCapacity( );

 public:
    stack(int cap = DEF_CAPACITY  ): theArray( new Object[cap] ), capacity(cap),
                                     topOfStack(-1){ }
    Stack(const stack & rhs);
    ~Stack( );
    Stack & operator=(const Stack & rhs );
    bool empty( ) const { return  topOfStack == -1; }
    int size( ) const { return  topOfStack + 1; }

    Object & top( );
    const Object& top( )const;

    void push (const Object& x);
    void push (       Object&& x);
    void pop( );
```



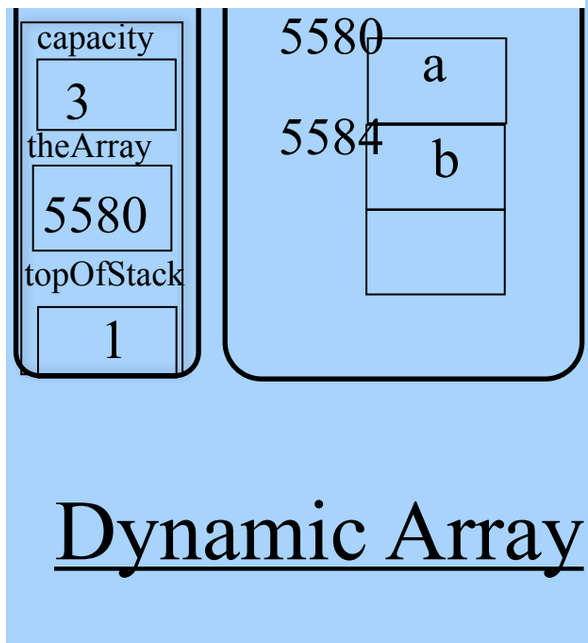


Dynamic Array

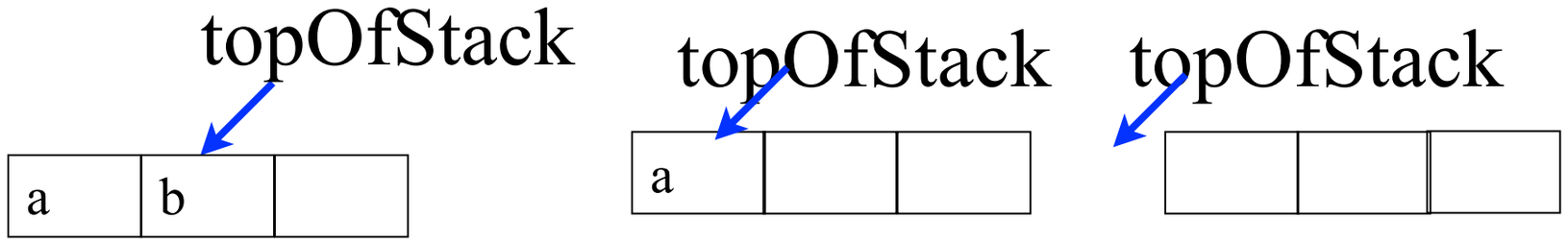
top



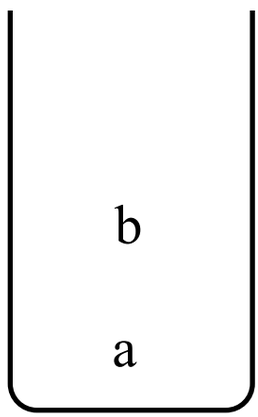

```
template<class Object>
Object & Stack<Object>::top( )
{
      if (empty())
          throw underflow_error("Top of empty stack");

     return theArray[topOfStack];
}
```



Dynamic Array



ADT



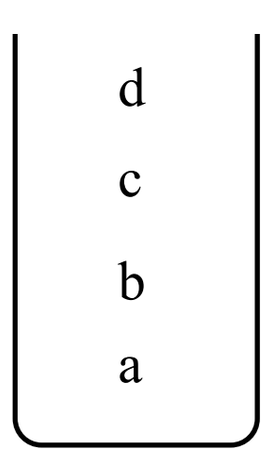
pop

```
template<class Object>
void Stack<Object>::pop( )
{
    if (empty())
        throw underflow_error("Pop from empty stack");
    --topOfStack;
}
```

pop( )

pop( )

ADT



push(‘a’)

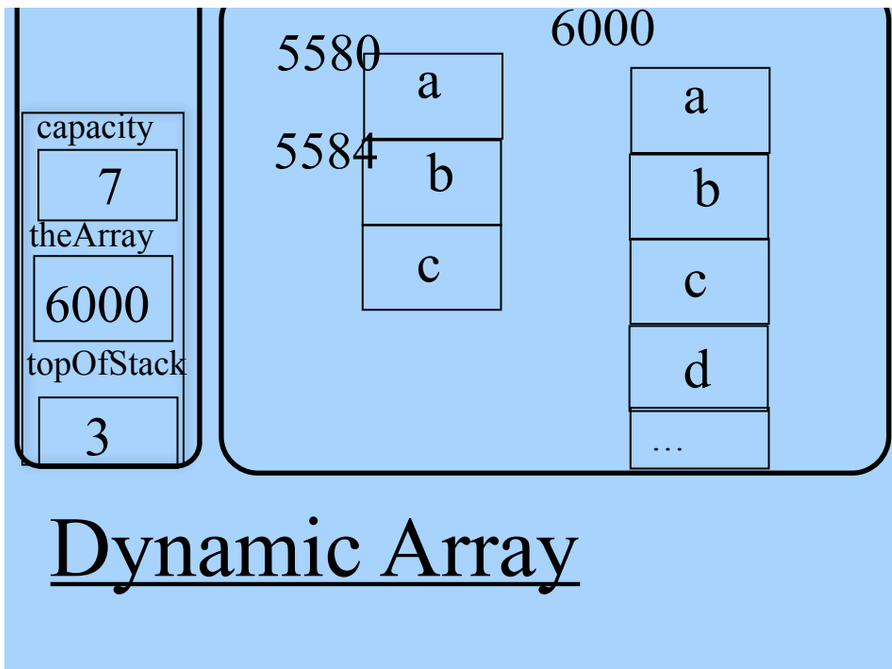
push(‘b’)

push(‘c’)

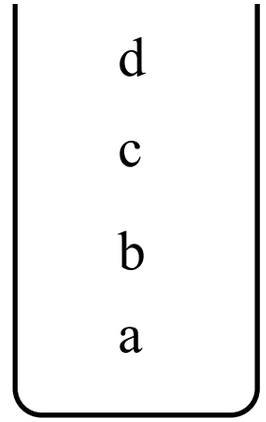
push(‘d’)

```
template<class Object>
void Stack<Object>::push( const Object & x )
{
    if ( ( topOfStack + 1 ) == capacity )
        increaseCapacity( );

    theArray[++topOfStack] = x;

}
```

```
template<class Object>
void Stack<Object>::increaseCapacity( )
{
    Object * oldArray = theArray;
    theArray = new Object[ capacity*2 + 1 ];

    for ( int i = 0; i < capacity; ++i )
        theArray[i] = std::move( oldArray[i] );

    capacity = capacity * 2 + 1;
    delete [ ] oldArray;
    oldArray = nullptr;
}
```



ADT



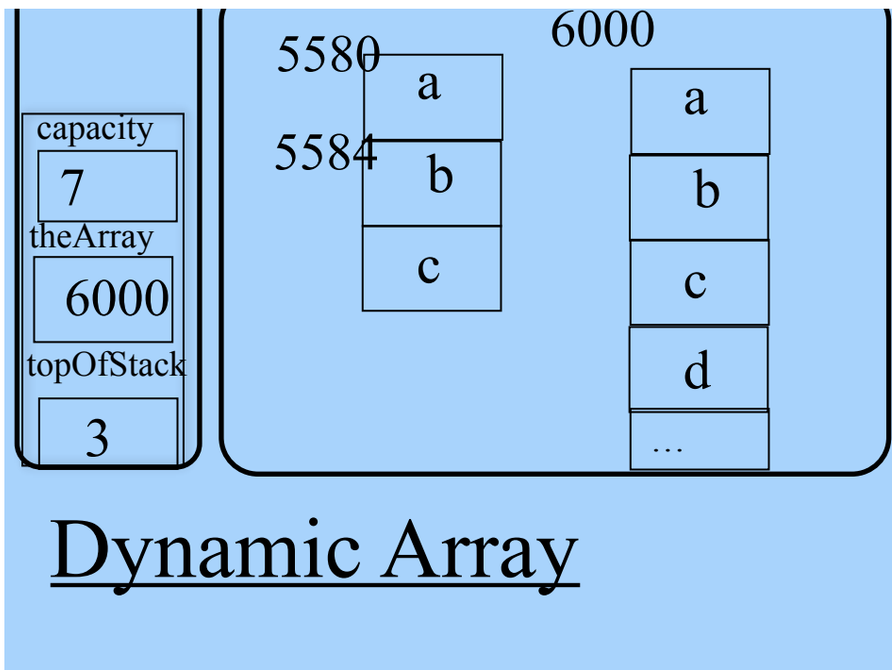

```
template<class Object>
void Stack<Object>::push( Object && x )
{
    if ( ( topOfStack + 1 ) == capacity )
        increaseCapacity( );

    theArray[++topOfStack] = std::move(x);

}
```

push(‘a’)

push(‘b’)

push(‘c’)

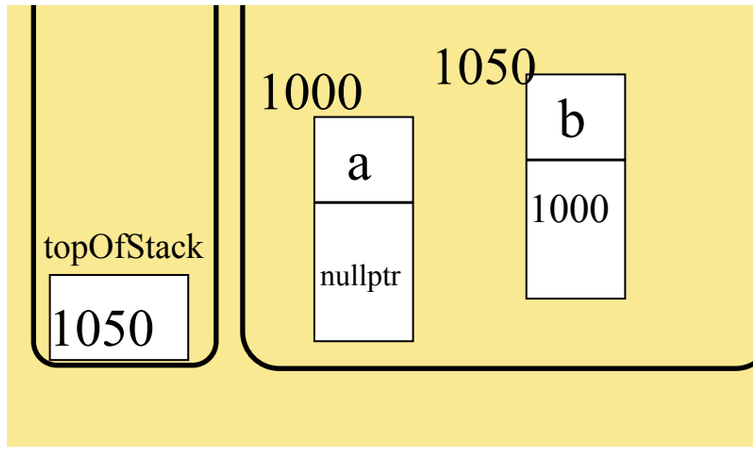
push(‘d’)



Stack Implementation 3: Singly Linked List

Uses:

```
struct ListNode
  {
     Object element;
     ListNode* next;
  }
```



- data members:

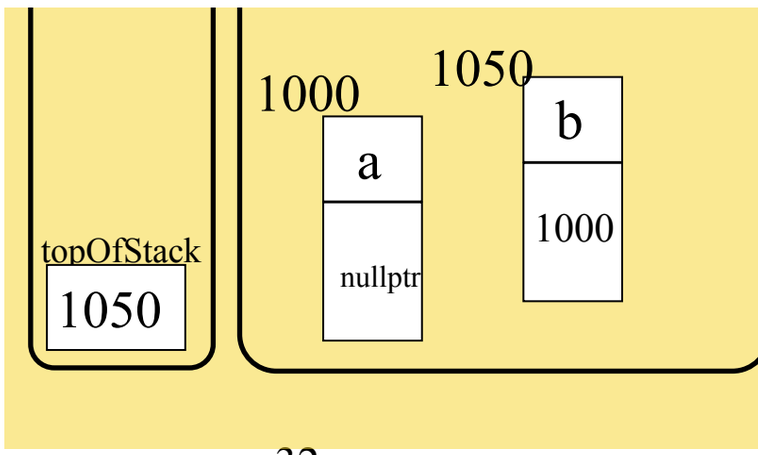
```
ListNode  *topOfStack;
```

- representation invariant: stack elements are stored in the list with bottom of the stack at end of the list and top of the stack at front of the list
- push/pop by inserting/removing at front of list
- All ops (except destructor) are O(1).

```
template <class Object>
class Stack
{
  public:
    Stack( );
    Stack( const Stack & rhs );
    ~Stack( );


    bool empty( ) const;
    const Object & top( ) const;
    void pop( );
    void push( const Object & x );
    const Stack & operator=( const Stack & rhs );

  private:
    struct ListNode
    {
        Object    element;
        ListNode *next;
        ListNode( const Object & theElement, ListNode * n = nullptr )
          : element( theElement ), next( n ) { }
        ListNode( Object && theElement, ListNode * n = nullptr )
          : element( std::move( theElement) ), next( n ) { }
     };
    ListNode *topOfStack;
};
```

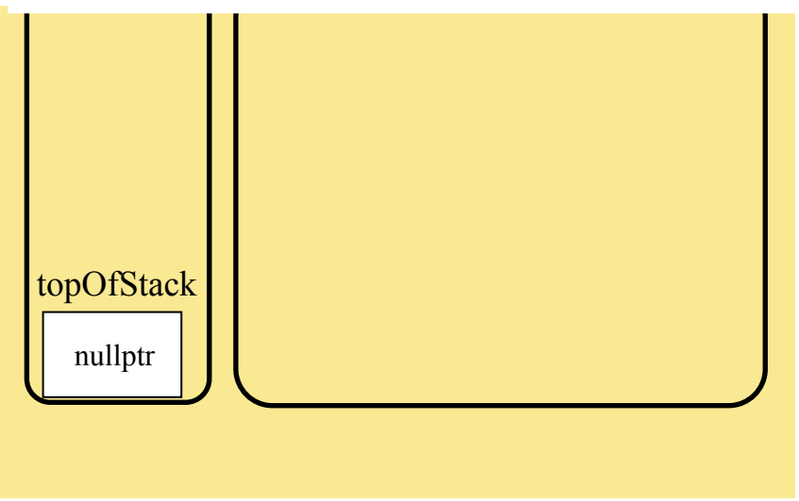
This code has been modified from the book code to be closer to the c++ implementation of the stack container adapter

notice constructor for the ListNode class



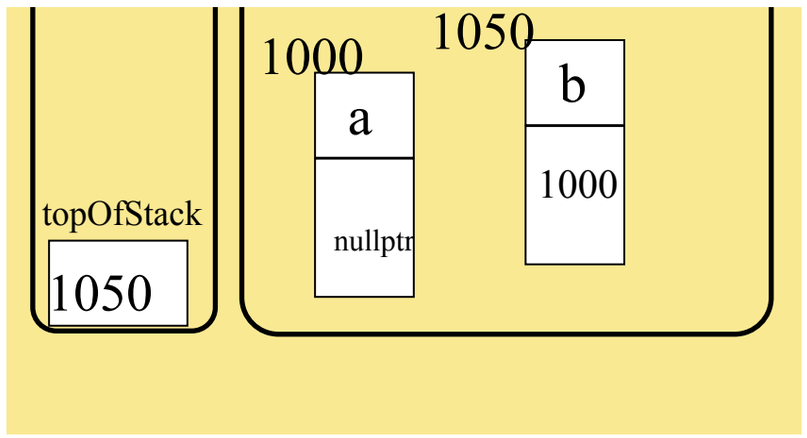
Constructor

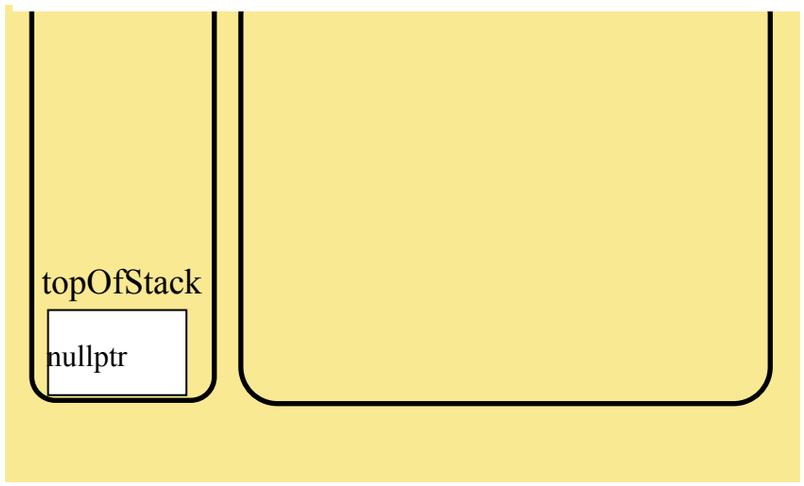
```
// Construct the stack.
template <class Object>
Stack<Object>::Stack( )
{
    topOfStack = nullptr;
}
```



empty

```
// Test if the stack is logically empty.
// Return true if empty, false, otherwise.
template <class Object>
bool Stack<Object>::empty( ) const
{
    return topOfStack == nullptr;
}
```

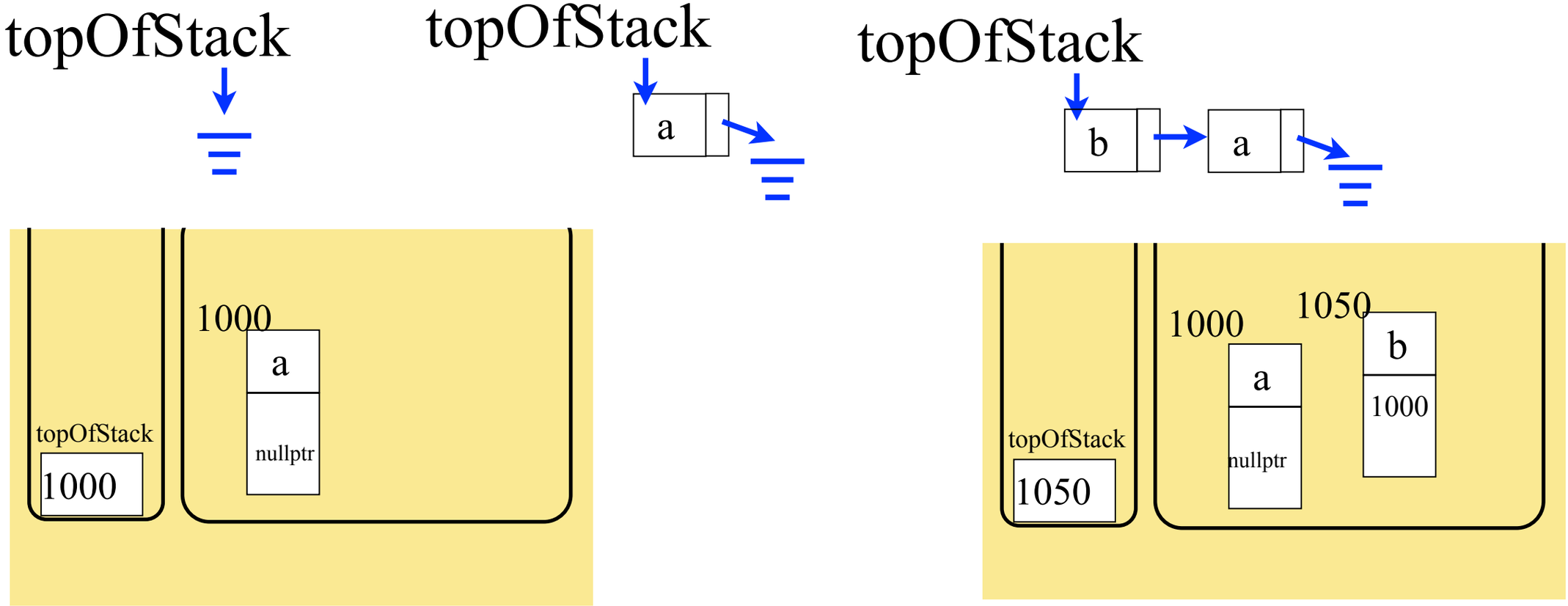




How would we write push(Object && x)?

push

```
// Insert x into the stack.
template <class Object>
void Stack<Object>::push( const Object & x )
{
    topOfStack = new ListNode( x, topOfStack );
}
```

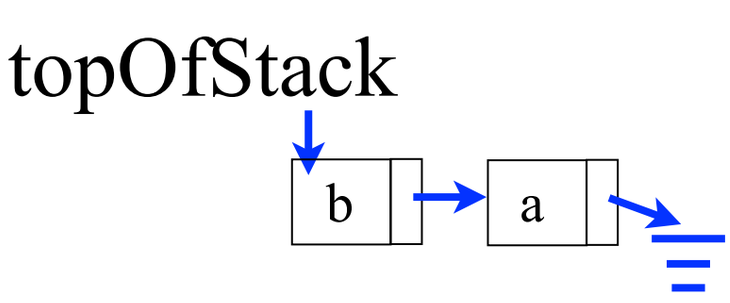


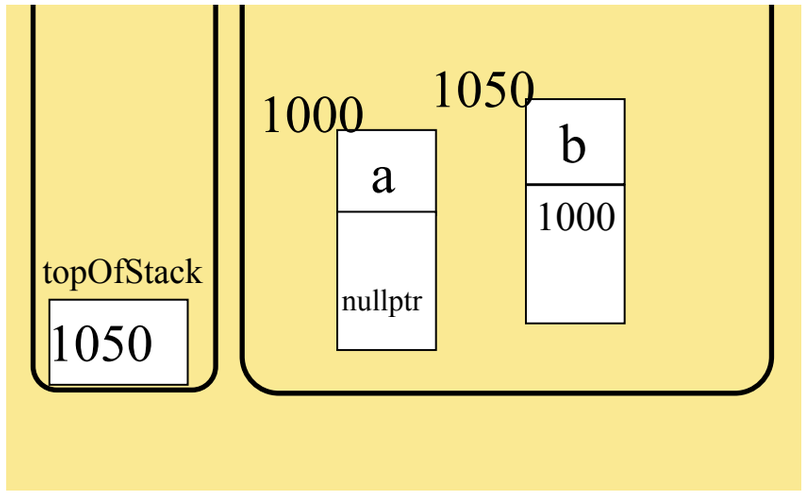
push

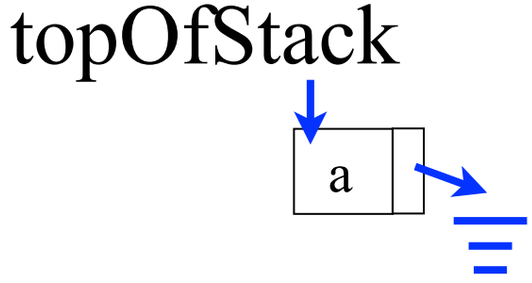
```
// Insert x into the stack.
template <class Object>
void Stack<Object>::push( Object && x )
{
    topOfStack = new ListNode( std::move( x ), topOfStack );
}
```

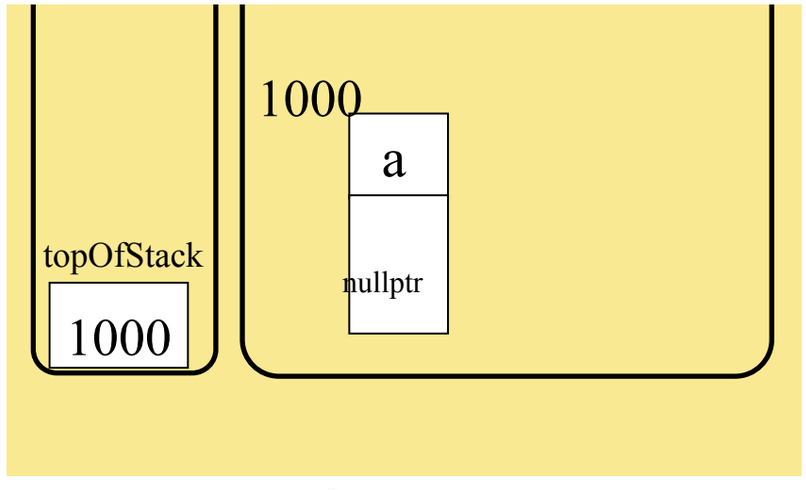
pop

```
// Insert x into the stack.
template <class Object>
void Stack<Object>::pop( )
{
    if ( empty( ) )
        throw  underflow_exception("Pop from empty stack");
     ListNode *oldNode = topOfStack;
     topOfStack = topOfStack->next;
     delete oldNode;
     oldNode = nullptr
}
```



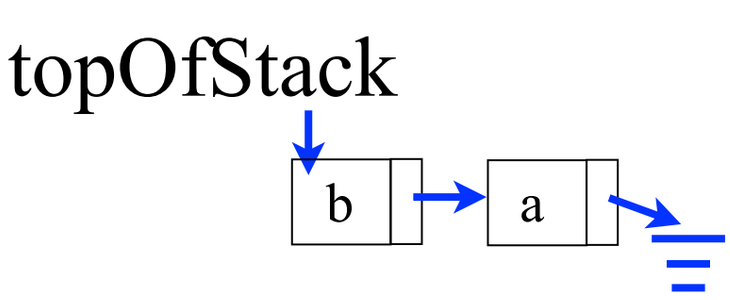


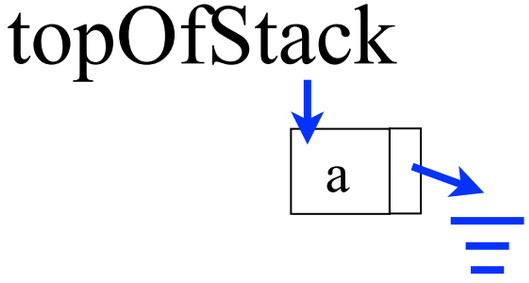


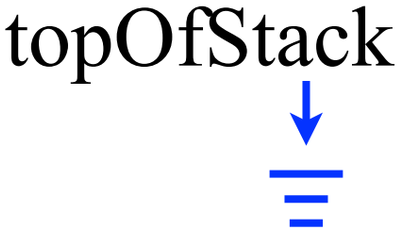


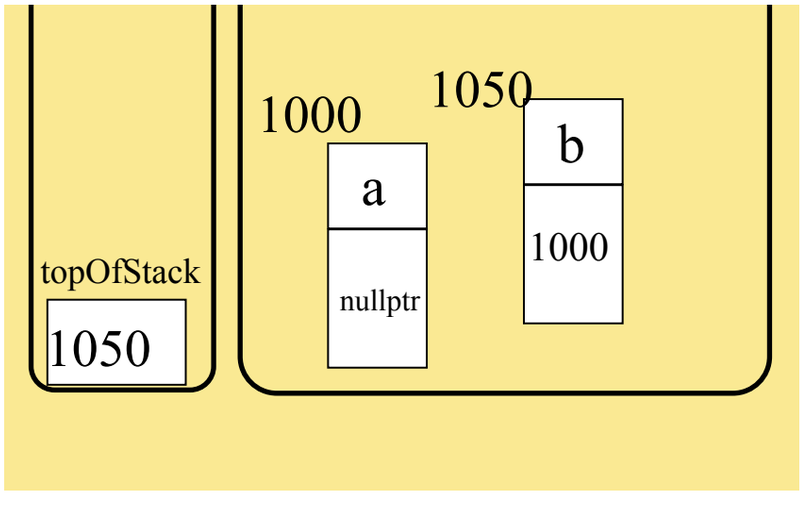
~Stack

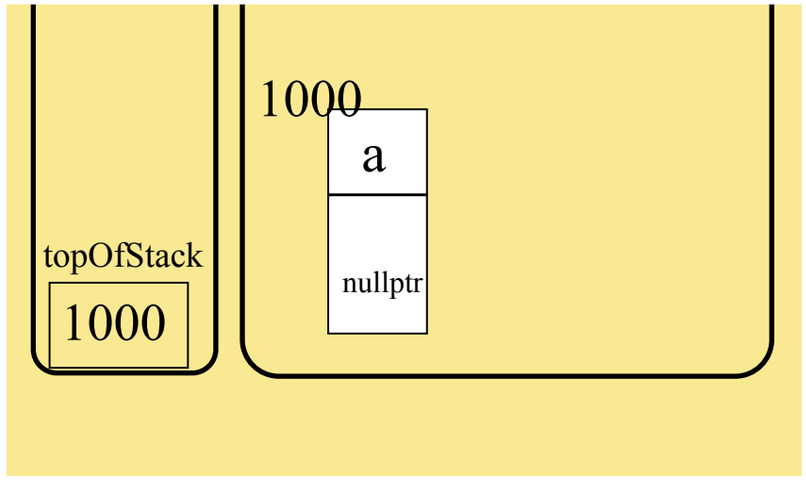
```
// Destructor.
template <class Object>
Stack<Object>::~Stack( )
{
    while( !empty( ) )
        pop( );

}
```











STL stack

#include<stack>

Elements are pushed/popped of the end

Adapter – stack<ItemType,ContainerType>

- bool empty( ); O(1)
- size_t size( ); O(1)
- ItemType top( ); O(1)
- void push(const ItemType& x); O(1)
- void push(ItemType&& x); O(1)
- void pop( ); O(1)

Notice that pop() returns void

A stack is used in the evaluation of expressions in computer languages.
A stack is used to implement function calls.

Implementing the Stack Class as a Container Adapter

– STL implements with vector, list, or deque

```
template<class T, class C = deque<T> >
class std::stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit stack(const C & a = C()) : c(a){} // Inherit the constructor
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() const { return c.back(); }
    void push(const value_type& n) { c.push_back(n); }
    void push(value_type && n) { c.push_back(std::move(n) ); }
    void pop() { c.pop_back(); }
};
```

modified from https://en.wikipedia.org/wiki/Stack_(C++)