EECE Department

ELC 303-B

Stacks

Dr. Ahmed Khattab

Objectives

1. Understanding and applying the Stack ADT.
2. Applications of the Stack ADT.
3. Implementing a Stack using an array.
4. Implementing the Stack ADT as a C++ Class.
5. Implementing an ADT with C++ templates.
6. Implementing a Stack using a dynamic list.
7. Comparison of different stack Implementations.

Robot Navigation Problem

Lets suppose that we adopt the following basic strategy for the robot:

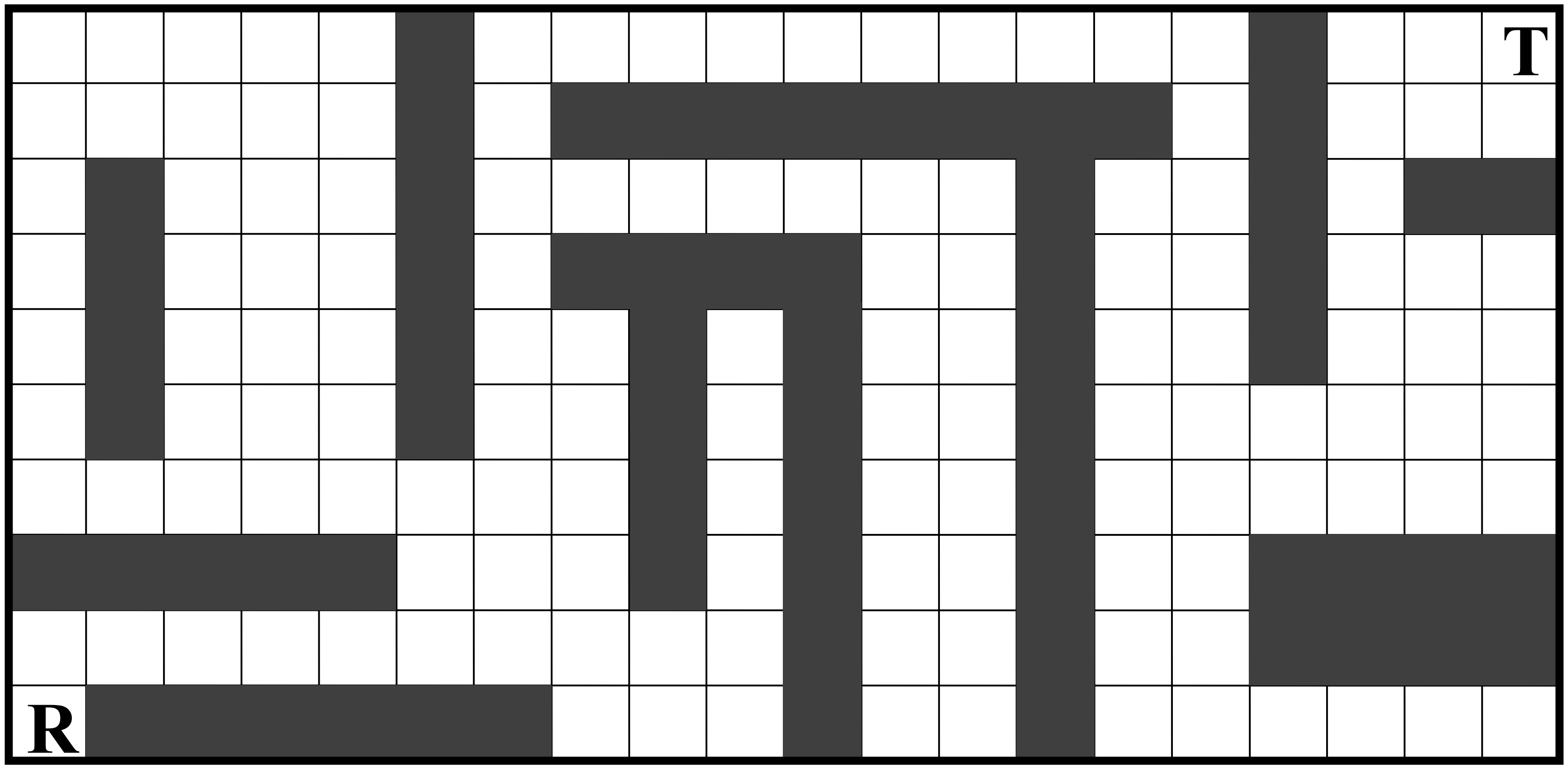
1. If possible, move in the direction of the target.

   If there are two directions both of which move the robot closer to the target, choose one arbitrarily.

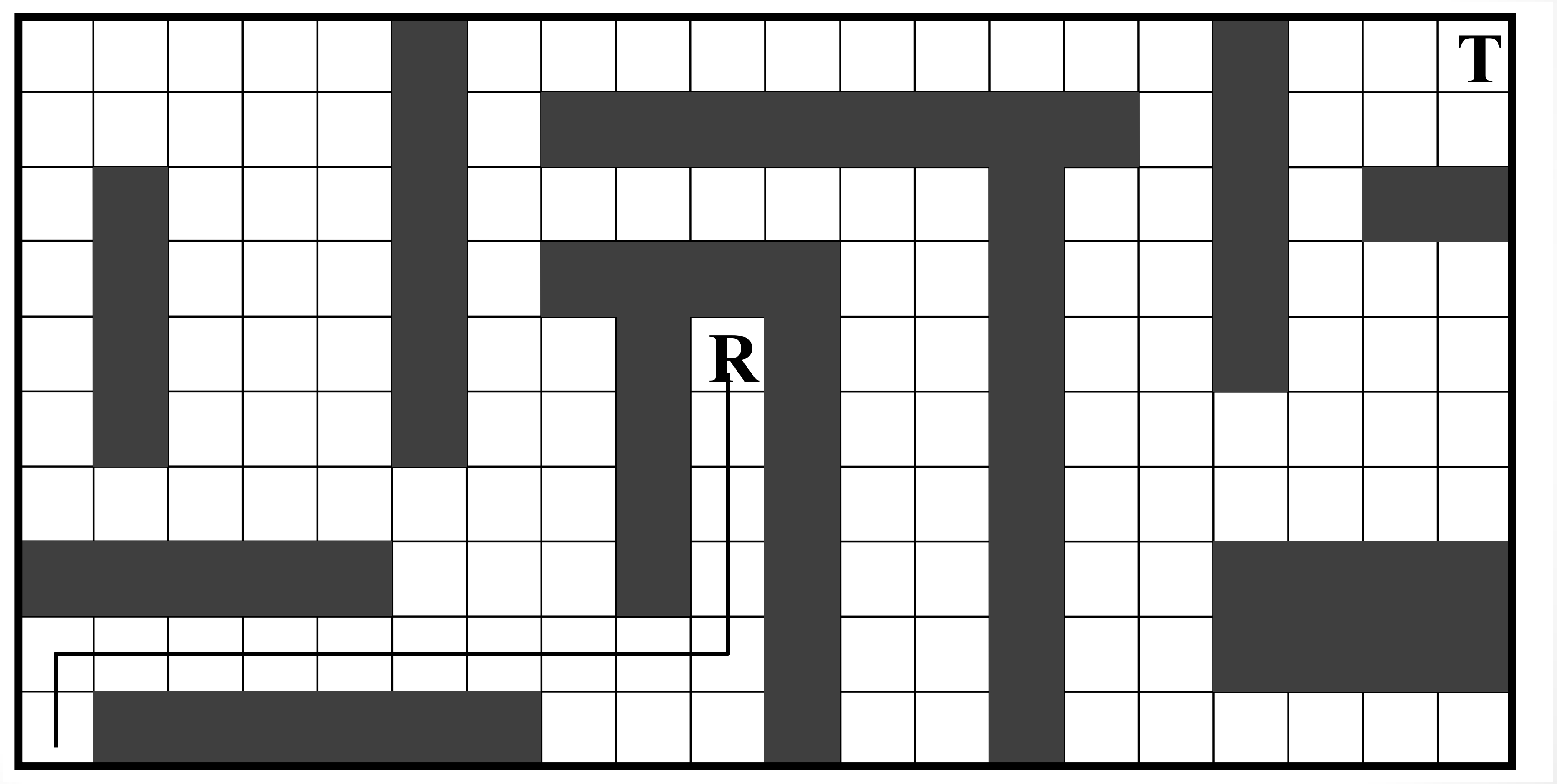
   (strategies may be fixed or arbitrary)

2. If the robot can’t move toward the target,
   try any other move arbitrarily.

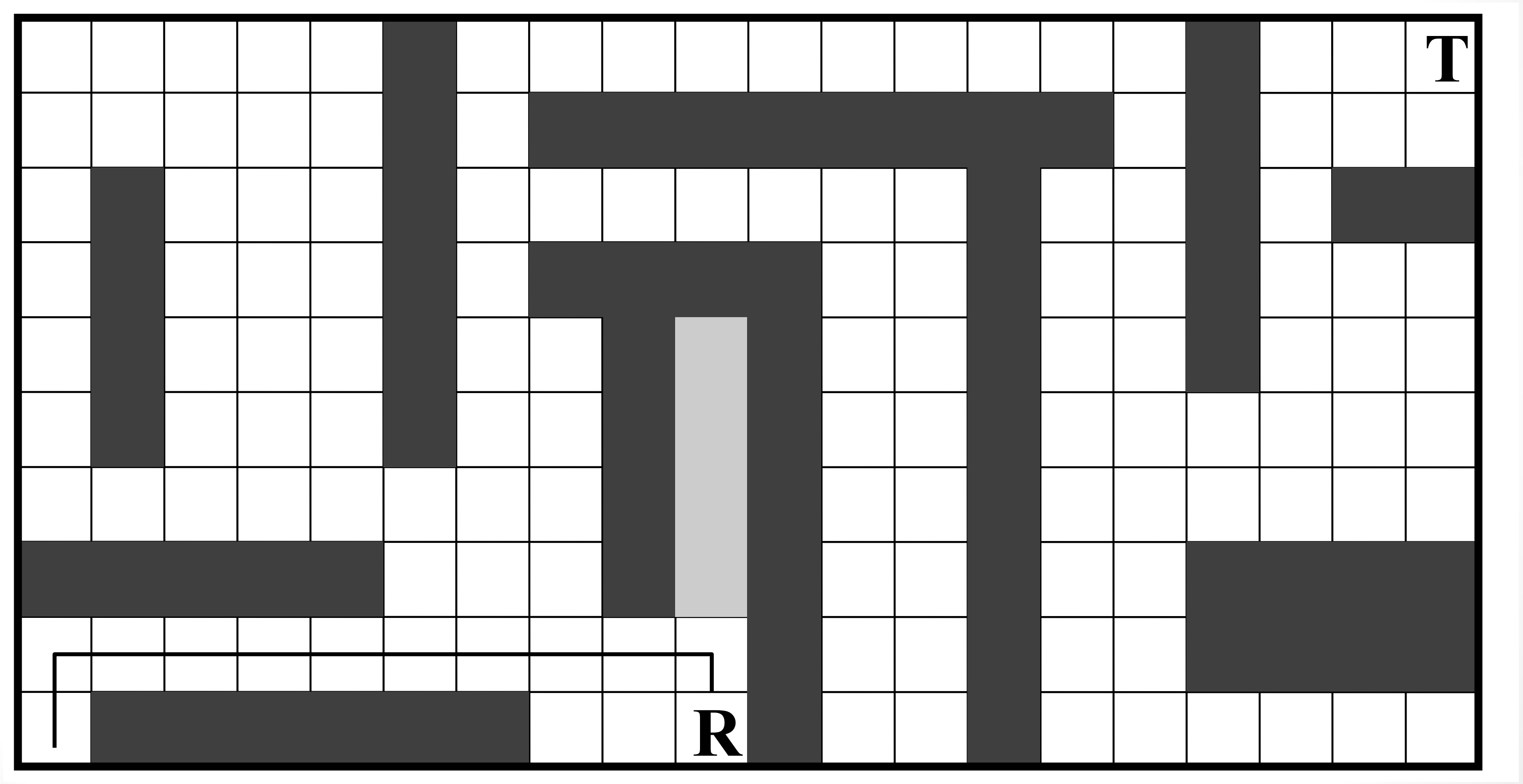
Initial conditions



The robot hits a dead end



backs up, tries new direction



Robot Navigation Problem

Based on our discussion above, we add the following rules to our algorithm:

3. As the robot moves, it marks the squares so that the robot doesn’t revisit them, except as indicated in rule 4.

4. When the robot reaches a dead end, it backs up through its previous positions until it finds an unexplored direction to try.

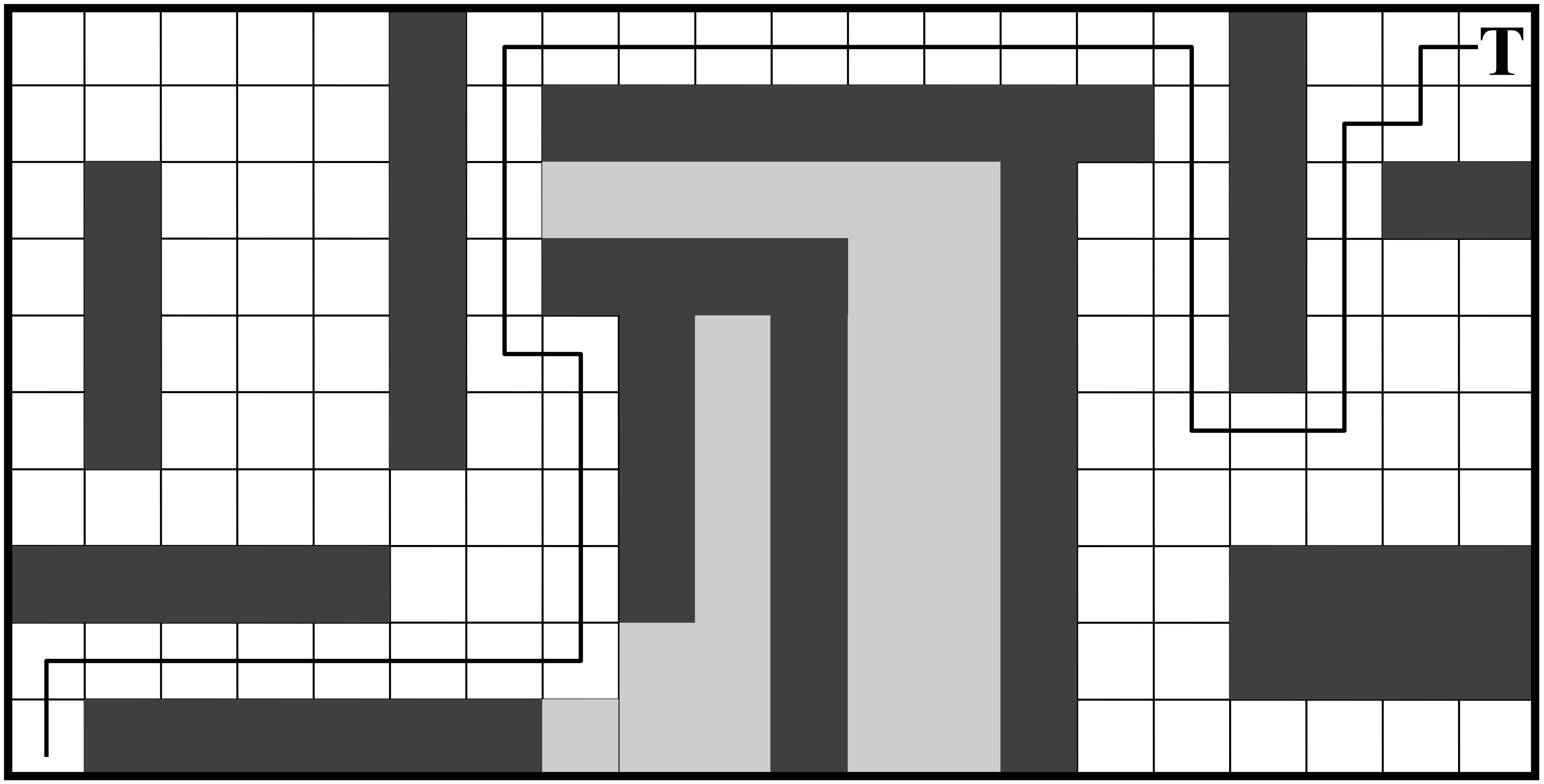
Backtracking

When base case is encountered we must backtrack

Go to the stack of previously visited squares

Pop them off one by one until a viable direction appears

Robot's path to the target



The stack

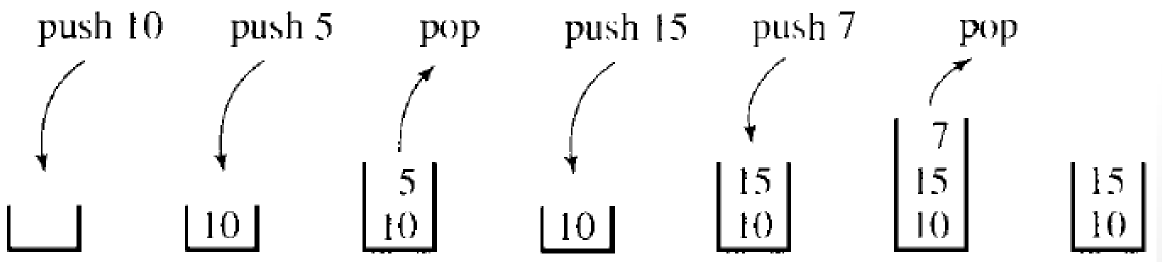
- The solution to this problem requires backtracking, which requires storing previously visited locations in reverse order

- The last one visited is the first one you backtrack to

- Such a structure is called a ‘stack’

Classic stack example

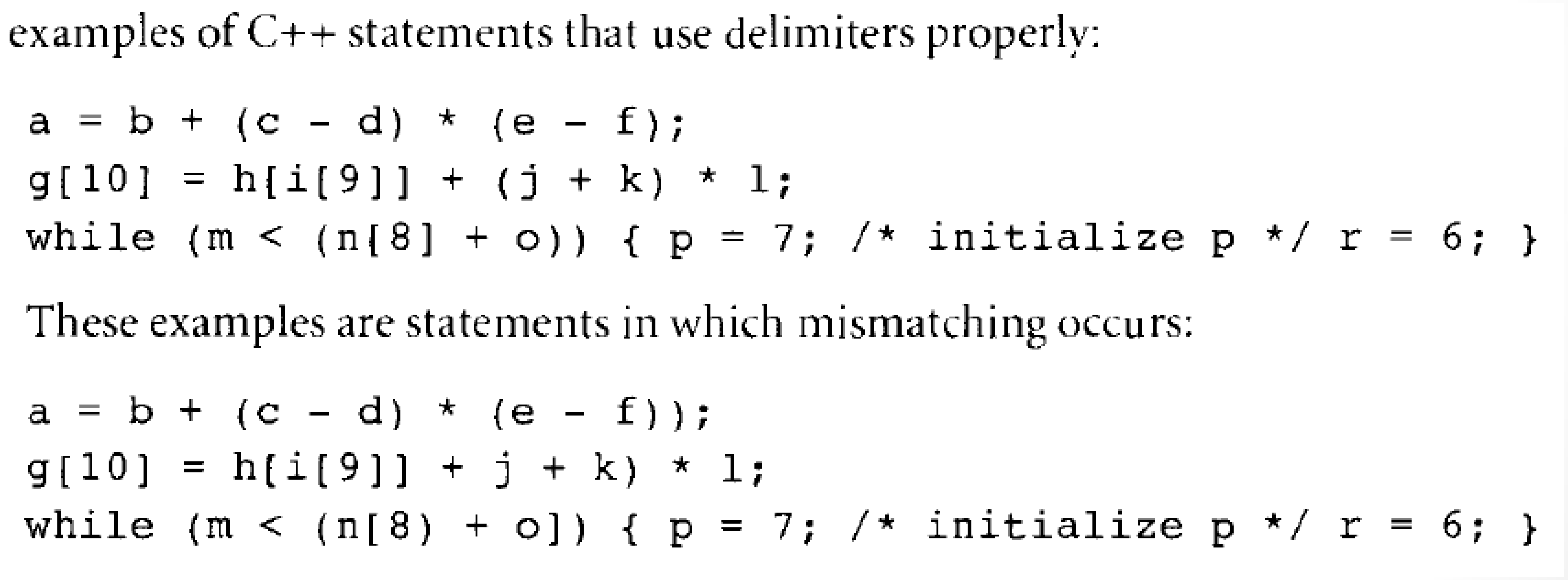
- Cafeteria trays, sheets of paper in a laser printer, etc.
- LIFO
- You 'push' items on
- You 'pop' items off
- Must always know where the top of the structure is
- Must know if the stack is empty
- Sometimes useful to know if it is full



Other Stack Applications

- Reverse a word. (homework)
- "undo" mechanism in text editors
- Language processing, e.g. matching delimiter in a program.
  - In C++, delimiters include (),[],{}, and /* .....*/.
  - Mismatching delimiters indicates a code error.
- Adding large numbers
- Evaluating mathematical expressions

Matching Delimiter



examples of C++ statements that use delimiters properly:

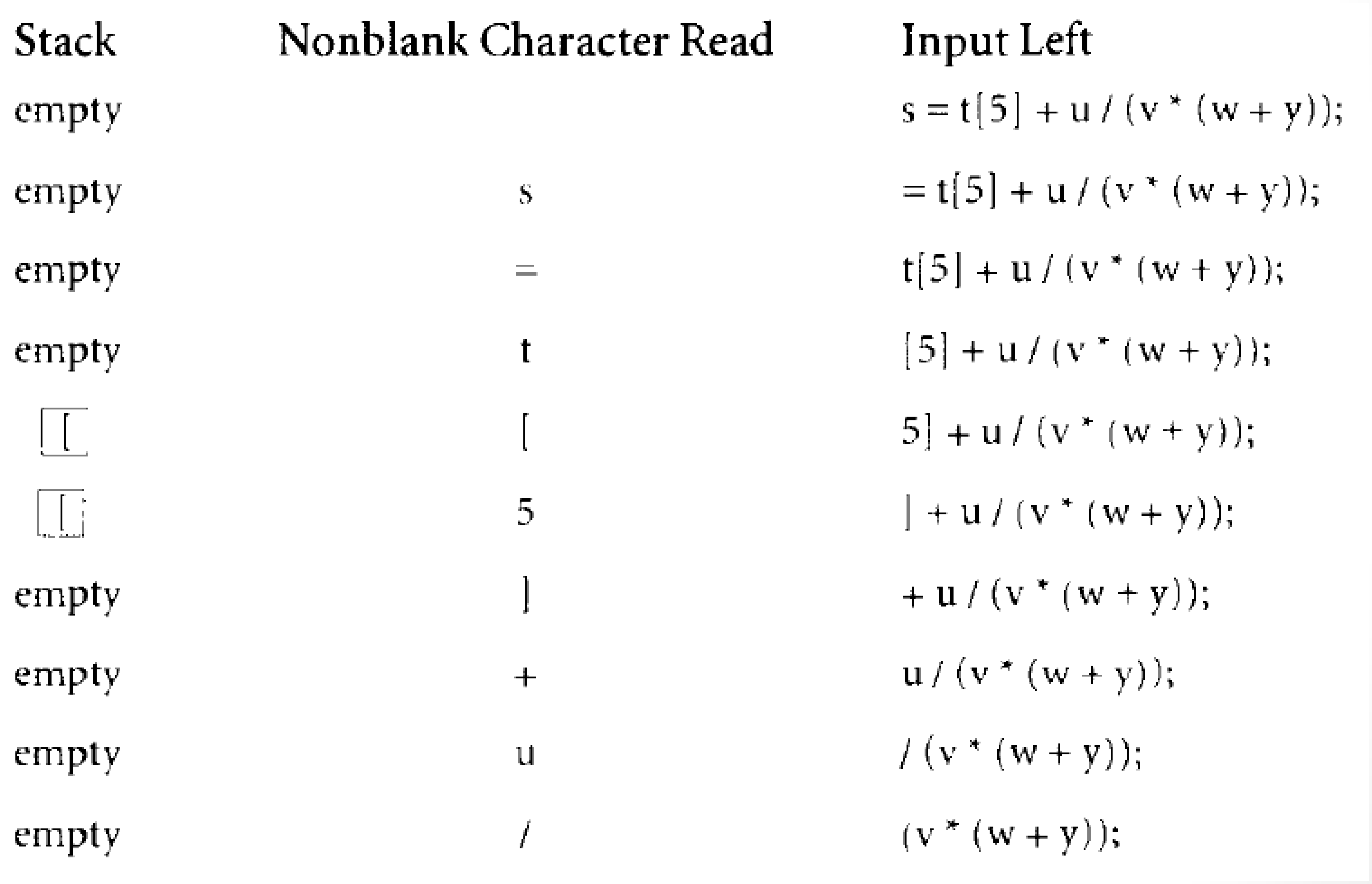
```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
```

These examples are statements in which mismatching occurs:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

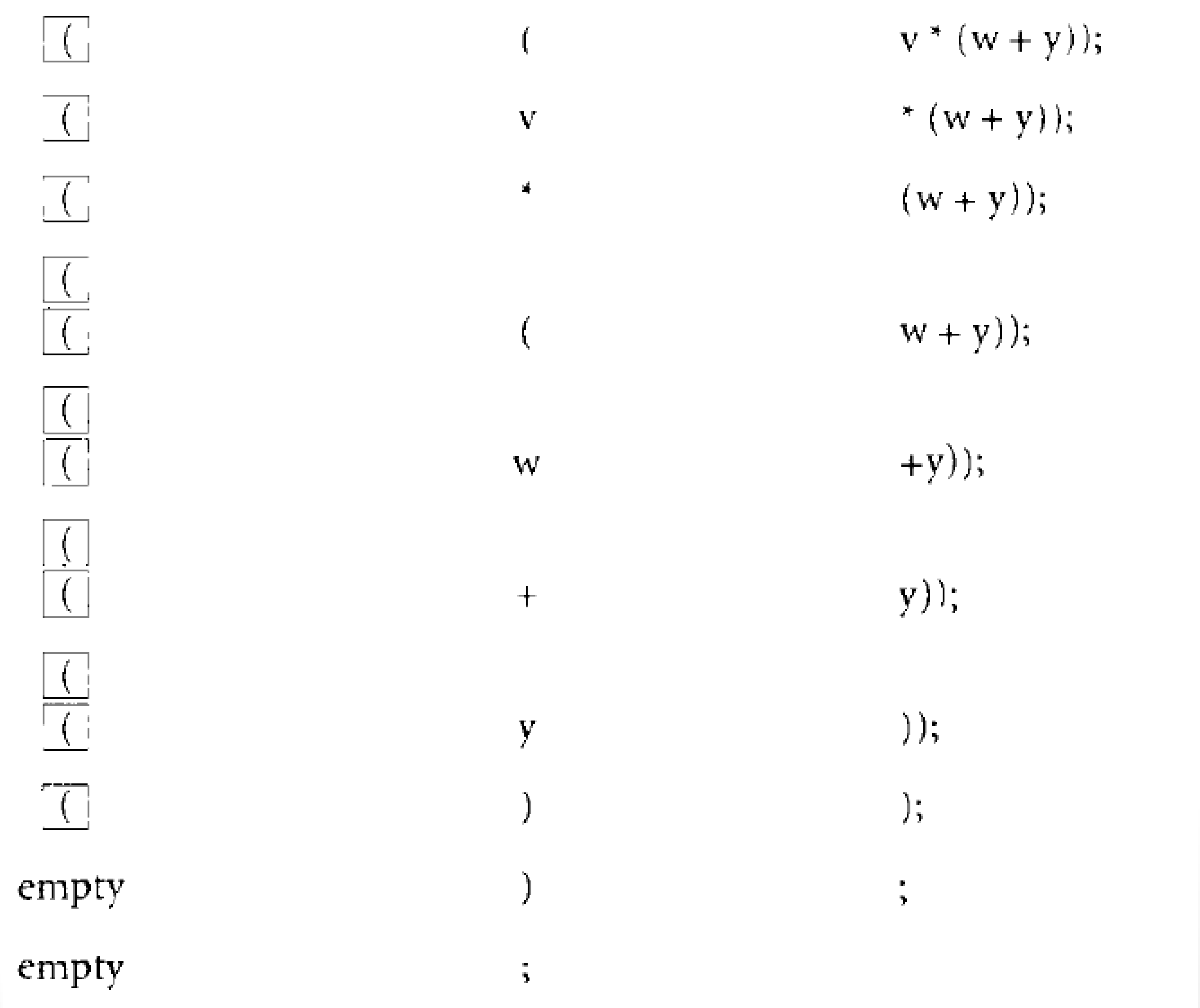
```
delimiterMatching(file)
  read character ch from file;
   while not end of file
      if ch is '(', '[', or '{'
           push(ch);
      else if ch is '/'
           read the next character;
           if this character is '*'
                push(ch);
           else ch = the character read in;
                continue; // go to the beginning of the loop;
      else if ch is ')', ']', or '}'
           if ch and popped off delimiter do not match
              failure;
      else if ch is '*'
           read the next character;
           if this character is '/' and popped off delimiter is not '/'
                failure;
           else ch = the character read in;
                push back the popped off delimiter;
                continue;
    // else ignore other characters;
        read next character ch from file;
   if stack is empty
        success;
   else failure;
```

Example: Processing `s=t[5]+u/(v*(w+y));`



Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));

Example: Processing `s=t[5]+u/(v*(w+y));`

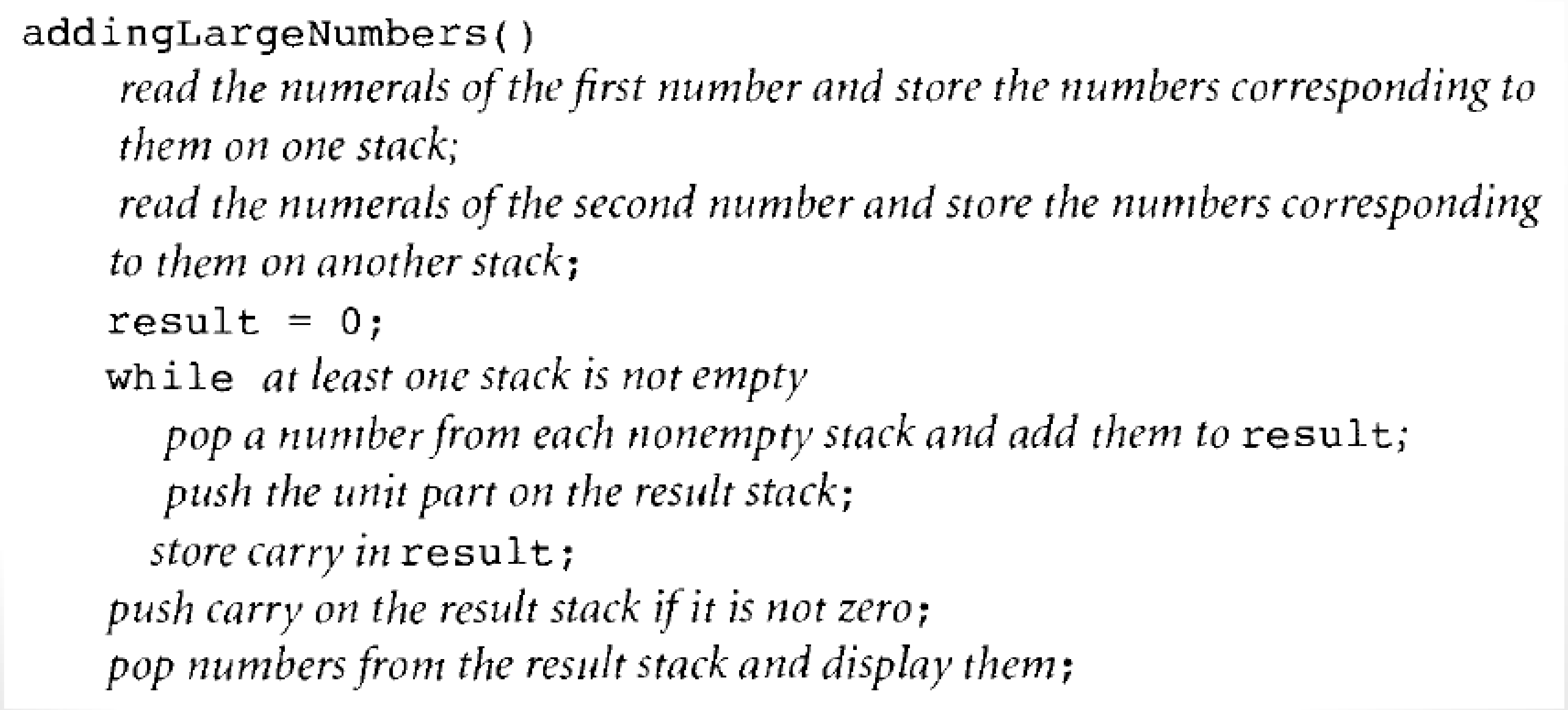


$\boxed{(}$	(	v * (w + y));
$\boxed{(}$	v	* (w + y));
$\boxed{(}$	*	(w + y));
$\boxed{(}$ $\boxed{(}$	(	w + y));
$\boxed{(}$ $\boxed{(}$	w	+y));
$\boxed{(}$ $\boxed{(}$	+	y));
$\boxed{(}$ $\boxed{(}$	y	));
$\boxed{(}$	)	);
empty	)	;
empty	;	

Adding Large Numbers

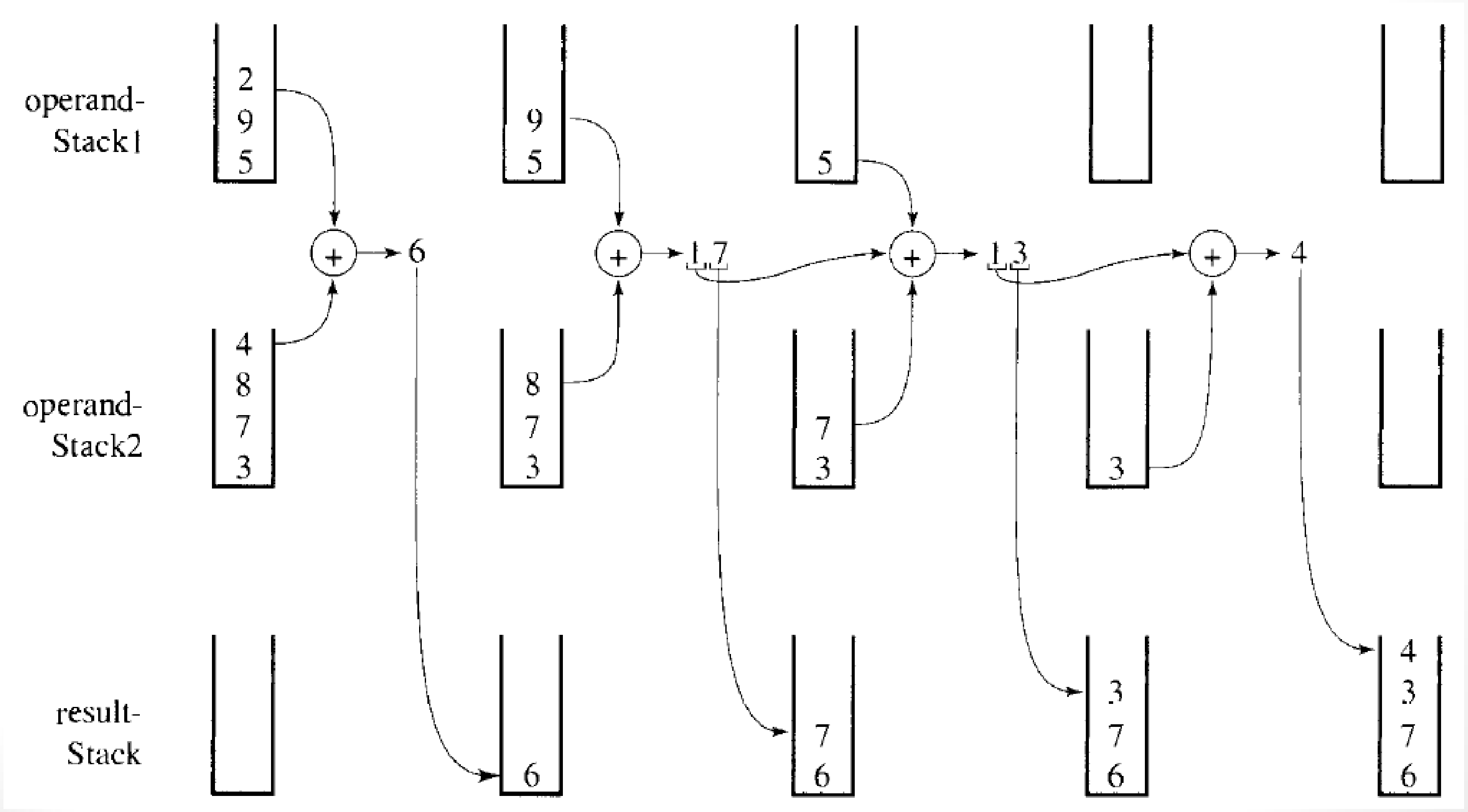
- Add the following 2 numbers:

  18,274,364,583,929,273,748,459,595,684,373
  8,129,498,165,026,350,236,



```
addingLargeNumbers()
    read the numerals of the first number and store the numbers corresponding to
    them on one stack;
    read the numerals of the second number and store the numbers corresponding
    to them on another stack;
    result = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to result;
        push the unit part on the result stack;
        store carry in result;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Example: Adding Large Numbers using Stacks



Stack Operations

- **Push operation**
  - adds a new item to the top of the stack, or initializes the stack if it is empty.
  - If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state.
- **Pop operation**
  - removes an item from the top of the stack.
  - if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed).
- **Clear operation** removes all stack elements
- **isEmpty** checks if the stack has data or not.
- **top operation** (also known as peek) returns the value of the first element without removing it.

Stack ADT (Abstract Data Type)

Characteristics:

- A stack S stores items of some type (stackElementType) in Last-In, First-Out (LIFO) order.

Operations:

stackElementType S.pop()

Precondition: !S.isEmpty()

Postcondition: S = S with top removed

Returns: The item x such that S.push(x) was the most recent invocation of push.

Stack ADT: push() and top()

void S.push(stackElementType)

Precondition: None

Postcondition: Item x is added to the stack, such that a subsequent S.pop() returns x.

stackElementType S.top()

Precondition: ! S.isEmpty()

Postcondition: None

Returns: The item x such that S.push(x) was the most recent invocation of push.

Stack ADT: isEmpty()

bool S.isEmpty()

Precondition:	None
Postcondtion:	None
Returns:	true if and only if S is empty, i.e., contains no data items.

Stacks and lists

- Stacks share many of the characteristics of lists, except they have a restriction on where data must be accessed or stored (only at the top)
- Data in stacks is homogeneous
- Use list-type implementations

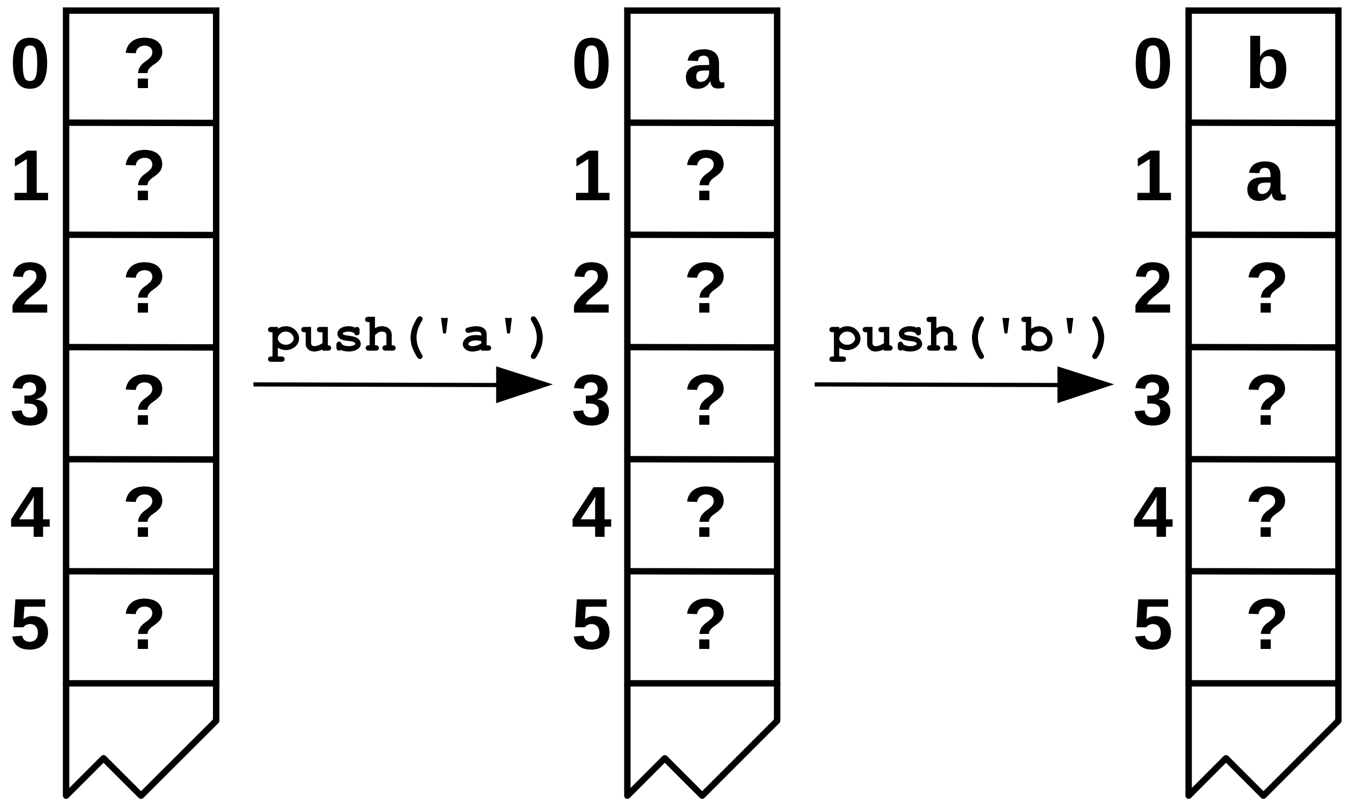
Ways to implement stacks

- Arrays
- Linked lists (dynamic stacks)

Array implementations

- What is wrong with the following example?

Array-based stack (version 1)



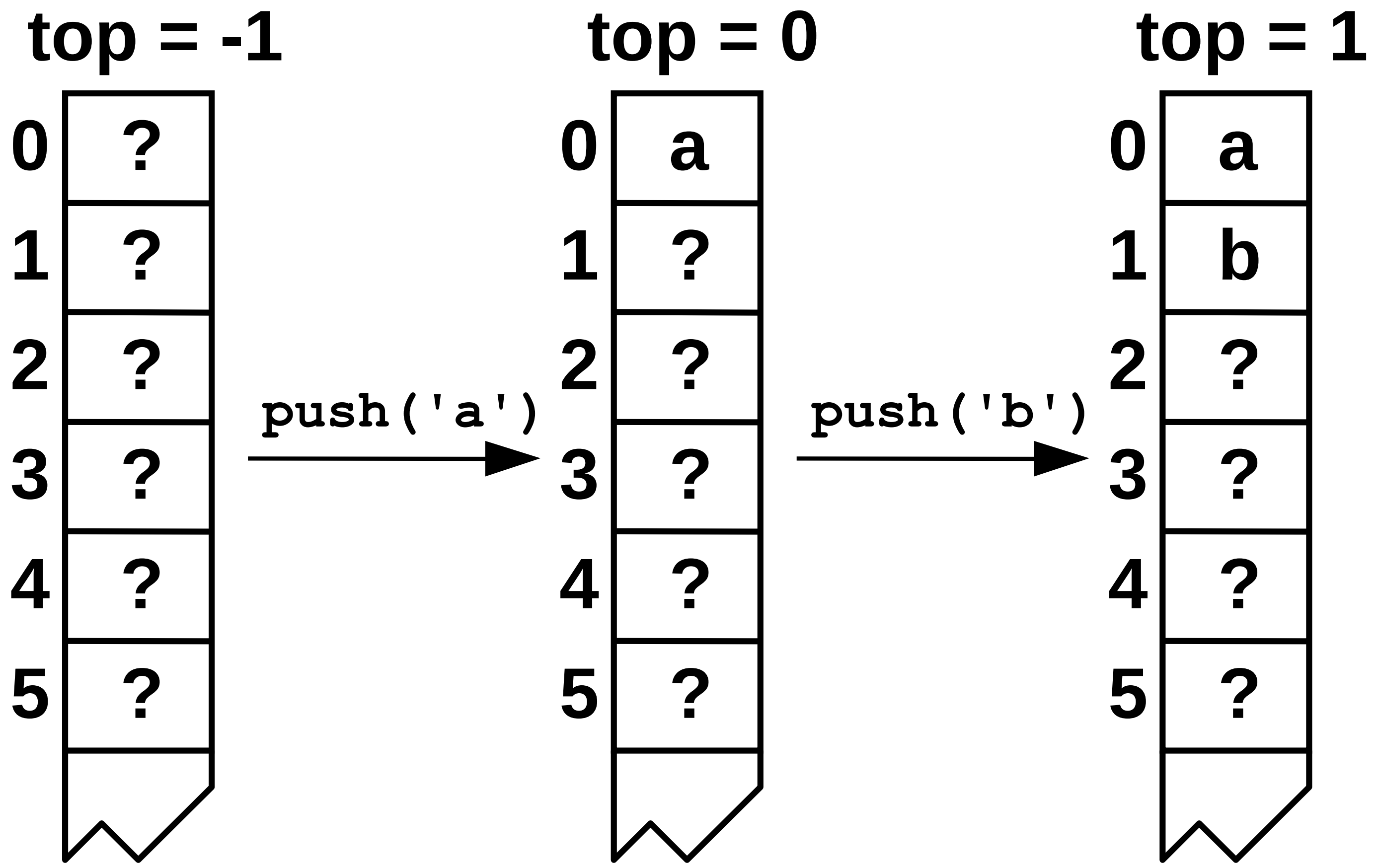
Problems

- The top is fixed, therefore all access for storage (push) or retreival (pop) must go through it.
- To keep it LIFO, the most recent item must be on top
- As new items are added older items must shuffle down.
- This is slow and unnecessary

A better implementation

- Instead of fixing the top and moving the data, fix the data and move the top
- As long as we know what top is we can still do all push and pop functions associated with LIFO

Version 2: floating top



Code Example 8-1

```
// Code Example 8-1: Stack implemented using an Array

const int maxStackSize = 1000;
typedef char StackElementType;
```

Class Stack

```
class Stack {
public:
   Stack();
   void push(StackElementType item);
   StackElementType pop();
   StackElementType top();
   bool isEmpty();
private:
   StackElementType stackArray[maxStackSize];
   int topIndex;
};
```

Header File Coding Tip
Avoid Multiple Header Inclusion

```
#ifndef _FILE_NAME_H_   //  #ifndef – it stands for "if not defined"
#define _FILE_NAME_H_

/* code */

#endif //  #ifndef _FILE_NAME_H_
```

- By using the #ifndef directive, you can include a block of text only if a particular expression is undefined; then, within the header file, you can dene the expression. This ensures that the code in the #ifndef is included only the first time the file is loaded.

Stack constructor

```
// cx8-2.cpp
// Code Example 8-2: Implementation file, stack implemented using an Array

#include "cx8-1.h"

Stack::Stack()
{
   topIndex = -1;
}
```

push( ) and pop( ) methods

```
void Stack::push(StackElementType item)
{// ensure array bounds not exceeded
   assert(topIndex < maxStackSize-1);
    ++topIndex;
   stackArray[topIndex] = item;
}
StackElementType Stack::pop()
{  // ensure array bounds not exceeded
   assert(topIndex >= 0);
   int returnIndex=topIndex;
   --topIndex;
   return stackArray[returnIndex];
}
```

top( ) and isEmpty( ) methods

```
StackElementType Stack::top()
{
   // ensure array bounds not exceeded
   assert(topIndex >= 0);
   return stackArray[topIndex];
}

bool Stack::isEmpty()
{
   return bool(topIndex == -1);
}
```

Other Variants

- Instead of fixing the maximum stack size using

  const int maxStackSize = 1000;

  one can initialize the maximum stack in the constructor (how?)
  - Gives more freedom in using the stack ADT

- A destructor ~Stack ( ) can be used to de-allocate the used space

  delete [] stackArray;

- Different other ways to implement **Push** and **Pop**
  - e.g., they can be made as bool to return whether or not the operation was successful instead of using assert()
  - Call be reference can be used to pass/get the stack value

Templates

- Templates are C++ tools that can be used to create generic classes
- Regular stack classes consist of one designated data type (from last example):

```
 const int maxStackSize = 1000;
 typedef char StackElementType;

private:
  StackElementType stackArray[maxStackSize];
```

Templates (continued)

- To use the previous code with a stack consisting of an array of ints, instead of chars, we would have to rewrite portions of it.
- A stack template would not have to be rewritten. The original class definition would be generic, allowing us to later declare a stack of any designated type. Example:

-
  ```
  int main()
  {       Token t;
          Stack < double > s;
          double op1, op2;
          bool done(false);
  ```

Pros and cons of templates

- Templates are not necessary if your ADT
  - Addresses only a specific client need
  - Is not intended for reuse

- Templates are necessary if your ADT is
  - Intended to be a general solution
  - Intended to be reused
  - Intended to work with any client

template syntax

- ‘template <class YourType>’ becomes part of the class definition and must also become the leading part of every member function.

Stack template class

```
// cx8-3.h
// Code Example 8-3: Stack declaration rewritten as a templated class

const int maxStackSize = 1000;
```

Code Example 8-3

```
template < class StackElementType >
class Stack {
public:
   Stack();
   void push(StackElementType item);
   StackElementType pop();
   StackElementType top();
   bool isEmpty();
   bool isFull();
private:
   StackElementType stackArray[maxStackSize];
   int topIndex;
};
```

Template constructor

```
// Code Example 8-4: Definition of Stack member functions using
   templates

#include "cx8-3.h”

template < class StackElementType >
Stack < StackElementType >::Stack()
{
  topIndex = -1;
}
```

Stack template push( )

```
template < class StackElementType >
void  Stack < StackElementType >::push(StackElementType item)
{
// ensure array bounds not exceeded
   assert(topIndex < maxStackSize-1);
    ++topIndex;
   stackArray[topIndex] = item;
}
```

Stack template pop( )

```
template < class StackElementType >
StackElementType Stack < StackElementType >::pop()
{
   // ensure array bounds not exceeded
   assert(topIndex >= 0);
   int returnIndex(topIndex);
   --topIndex;
   return stackArray[returnIndex];
}
```

Stack template top( )

```
template < class StackElementType >
StackElementType Stack < StackElementType >::top()
{
// ensure array bounds not exceeded
   assert(topIndex >= 0);
   return stackArray[topIndex];
}
```

isEmpty( ) and isFull( )

```
template < class StackElementType >
bool Stack < StackElementType >::isEmpty()
{
   return bool(topIndex == -1);
}


template < class StackElementType >
bool Stack < StackElementType >::isFull()
{
   return topIndex == maxStackSize - 1;
}
```

Stack Driver Example

- The following code represent a driver for the developed stack template.
  - The driver program begins by instantiating object doubleStack. This object is declared to be of class ***Stack< double > (pronounced "Stack of double")***. The compiler associates type double with type parameter in the class template to produce the source code for a Stack class of type double. *Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.*

```
// Stack class template test program.
#include <iostream>
using std : : cout ;
using std : : endl ;
#include "cx8-3.h" // Stack class template definition
#include "cx8-4.cpp" // Stack class template implementation

int main ( )
{
        Stack< double > doubleStack; // default size
        double doubleValue = 1 . 1 ;

        cout << "Pushing element s onto double Stack \n" ;

         // push doubles onto double Stack until it is full
        while ( !doubleStack.isFull () )
        {
                doubleStack.push (doubleValue);
                cout << doubleValue << '  ' ;
                doubleValue += 1 . 1 ;
        } // end while

        cout << "\n Stack is full . Cannot push " << doubleValue
            << "\n \n Popping elements from doubleStack \n" ;
```

```
// pop elements from doubleStack
while (!doubleStack.isEmpty () )
{
            doubleValue = doubleStack . pop ();
            cout << doubleValue << ' ' ;
}

cout << "\n Stack is empty . Cannot pop\n" ;

Stack< int > intStack ; // default size
int intValue = 1 ;
cout << "\n Pushing element s onto intStack \n" ;

// push integers onto intStack
while ( !intStack . isfull () )
{
            inStack . push(intValue++)
            cout << intValue << ' ' ;
} // end while

cout << "\ n Stack is full . Cannot push " << intValue
     << "\n \n Popping element s from intStack \n" ;
```

```
        cout << "\n  Stack is full. Cannot push " << intValue
             << "\n \n Popping element s from intStack \n" ;

        // pop elements from intStack
        while ( !intStack . Is Empty () )
                  cout << int(intStack . pop()) << ' ' ;

        cout << "\n Stack is empty. Cannot pop" << endl ;
        return 0 ;

} // end main
```

Linked-list implementation

- Array-based stacks have the disadvantage that they are static structures
- They may waste space
- They may not have enough space for elements

Dynamic stacks

- Do not have unused space
- Do have extra memory requirements (for pointers)
- Do not run out of space, unless there is no more room on the heap

List-based stack, public

```
template < class StackElementType >
class Stack {
public:
  Stack();
  void push(StackElementType e);
  StackElementType pop();
  StackElementType top();
  bool isEmpty();
```

The private section

- Must include a structure definition or be based on another class that defines the node object
- Similar to linked lists

List-based stack, private

```
private:
  struct Node;
  typedef Node * Link;
  struct Node {
    StackElementType data;
    Link next;
  };
  Link head;
};
```

Constructor

```
template < class StackElementType>
Stack < StackElementType >::Stack()
{
   head = NULL;
}
```

Push

- Pushing a value onto the stack means having to create a new node, assign it the data value, and prepend it to the linked list
- The pointer to this node becomes the top of the stack (head of the list)

push

```
template < class StackElementType >
void
Stack < StackElementType >::push(StackElementType e)
{
   Link addedNode = new Node;
   assert(addedNode!=NULL);
   addedNode->data = e;
   addedNode->next = head;
   head = addedNode;
}
```

Comparing Stack Implementations

- The array implementation allocates the entire required space while linked implementation only occupies the actual required space.
  - However, note that the list node needs more space as it stores linking information in addition to the value information.
- The implementation choice depends on the stored information
  - Case A: Integer stack of max 100 Elements
    - Array Implementation requirements: 2Byte/item*100item + 2bytes*2indicies = 204Bytes (always reserved)
    - Linked Implementation: 6 bytes/ item (2 for value and four for next) + 4bytes for external pointers
    - The array implementation size is reached by linked implementation after 33 elements only.
  - What would be the case if the stored data has a bigger size, e.g. a string of length 100Bytes?

Chapter Summary

- The Stack ADT is a LIFO data structure characterized by the operations **push** and **pop**.
- The Stack ADT can be implemented using either an array, which has a fixed maximum size, or a linked list which can grow dynamically.
- Generic classes can be created using templates in C++.
- The Stack ADT can be used to organize the stack frames used at runtime to keep track of function calls, both recursive and non-recursive.