# Lecture #5

- Stacks
- Queues

# The Stack: A Useful ADT

A stack is an ADT that holds a collection of items (like ints) where the elements are always added to one end.

Just like a stack of plates, the last item pushed onto the top of a stack is the first item to be removed.

Stack operations:

- put something on top of the stack (PUSH)
- remove the top item (POP)
- look at the top item, without removing it
- check to see if the stack is empty

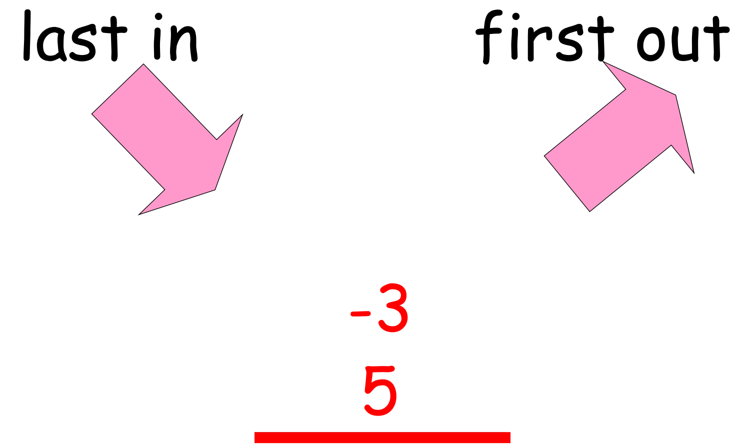
We can have a stack of any type of variable we like: ints, Squares, floats, strings, etc.

# The Stack

Note: The stack is called a Last-In-First-Out data structure.
Can you figure out why?

I can…

Push 5 on the stack.
Push -3 on the stack.
Push 9 on the stack.
Pop the top of the stack.
Look at the stack's top value.
Push 4 on the stack.
Pop the top of the stack
Pop the top of the stack
Look at the stack's top value.
Pop the top of the stack




Note: You can only access the top item of the stack, since the other items are covered.

# Stacks

```
class Stack   // stack of ints
{
public:
   Stack();          // c'tor
   void push(int i);
   int pop();
   bool is_empty(void);
   int peek_top();
private:
   ...
};
```

```
int main(void)
{
    Stack is;

    is.push(10);
    is.push(20);

    ...
}
```

**Question**:
What type of data structure can we use to implement our stack?

**Answer**:
How about an array and a counter variable to track where the top of the stack is?

# Implementing a Stack

```
const int SIZE  = 100;

class Stack
{
public:
    Stack()  {  m_top = 0;    }
    void push(int val)  {
        if (m_top >= SIZE) return; // ove
        m_stack[m_top] = val;
        m_top += 1;
    }

    int pop()  {
        if (m_top == 0) return -1; // un
        m_top -= 1;
        return m_stack[m_top];
    ...
private:
    int m_stack[SIZE];
    int m_top;
};
```

To initialize our stack, we'll specify that the first item should go in the $0^{th}$ slot of the array.

Let's make sure we never over-fill (overflow) our stack!

Place our new value in the next open slot of the array… m_top specifies where that is!

Update the location where our next item should be placed in the array.

We can't pop an item from our stack if it's empty! Tell the user!

Since m_top points to where our next item will be pushed…

Let's decrement it to point it to where the current top item is!

Extract the value from the top of the stack and return it to the user.

Let's use an array to hold our stack items.
This stack may hold a maximum of 100 items.

We'll use a simple int to keep track of where the next item should be added to the stack.

# Stacks

```
const int SIZE  = 100;

class Stack
{
public:
    Stack() { m_top = 0; }
    void push(int val) {
        if (m_top >= SIZE) re
        m_stack[m_top] = va
        m_top += 1;
    }

    int pop() {
        if (m_top == 0) return -1; // underflow
        m_top -= 1;
        return m_stack[m_top];
    }

    ...
private:
     int m_stack[SIZE];
     int m_top;
};
```

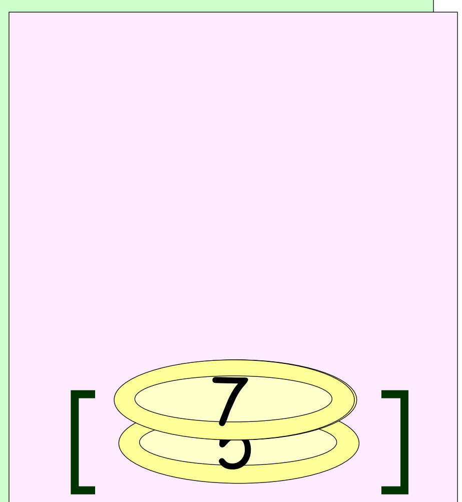
Currently, our m_top points to the next open slot in the stack…

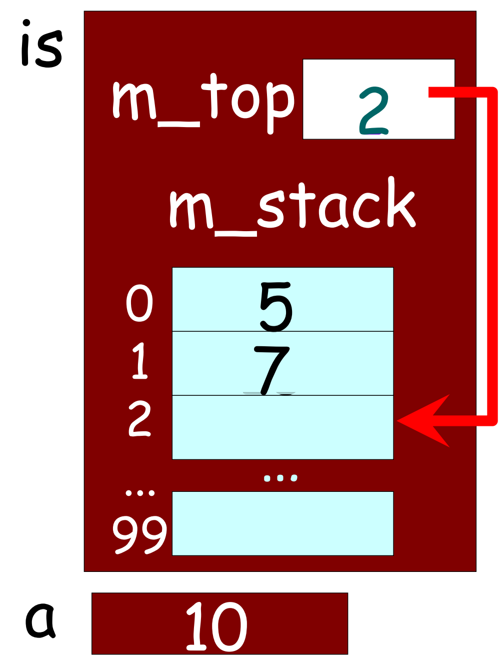
But we want to return the top item already pushed on the stack.

So first we must decrement our m_top variable…

```
int main(void)
{
     Stack is;
    int a;

     is.push(5);
     is.push(10);
     a = is.pop();
     cout << a;
     is.push(7);
}
```



```
const int SIZE  = 100;

class Stack
{
public:
   Stack() { m_top = 0; }
   void push(int val) {
      if (m_top >= SIZE) retu   ; // overflow
      m_stack[m_top] = val;
      m_top += 1;
   }

   int pop() {
      if (m_top == 0) return -1; // underflow
      m_top -= 1;
      return m_stack[m_top];
   }

   ...
private:
    int m_stack[SIZE];
    int m_top;
};
```

Always Remember:

When we push, we:

A. Store the new item in m_stack[m_top]

B. Post-increment our m_top variable

(post means we do the increment after storing)

Always Remember:

When we pop, we:

A. Pre-decrement our m_top variable

B. Return the item in m_stack[m_top]

(pre means we do the decrement before returning)

# Stacks

Stacks are so popular that the C++ people actually wrote one for you. It's in the Standard Template Library (STL)!

```
#include <iostream>
#include <stack>

int main()
{
  std::stack<int>  istack; // stack of ints

  istack.push(10);        // add item to top
  istack.push(20);

  cout << istack.top();   // get top value
  istack.pop();           // kill top value

  if (istack.empty() == false)
    cout << istack.size();
}
```

Here's the syntax to define a stack:

std::stack<type> variableName;

For example:

std::stack<string> stackOfStrings;
std::stack<double> stackOfDoubles;

So to get the top item's value, before popping it, use the top() method!

Note: The STL pop() command simply throws away the top item from the stack... but it doesn't return it.

# Stack Challenge

Show the resulting stack after the following program runs:

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

int main()
{
  stack<int>  istack;       // stack of ints

  istack.push(6);
  for (int i=0;i<2;i++)
  {
     int n = istack.top();
     istack.pop();
     istack.push(i);
     istack.push(n*2);
  }
}
```

# Stack Challenge

Show the resulting stack after the following program runs:

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

int main()
{
  stack<int>  istack;       // stack of ints

  istack.push(6);
  for (int i=0;i<2;i++)
  {
     int n = istack.top();
     istack.pop();
     istack.push(i);
     istack.push(n*2);
  }
}
```

# Common Uses for Stacks

Stacks are one of the most USEFUL data structures in Computer Science.

They can be used for:

- Storing undo items for your word processor
  The last item you typed is the first to be undone!
- Evaluating mathematical expressions
  5 + 6 * 3 → 23
- Converting from infix expressions to postfix expressions
  A + B → A B +
- Solving mazes

In fact – they're so fundamental to CS that they're built into EVERY SINGLE CPU in existence!

# A Stack… in your CPU!


Did you know that every CPU has a built-in stack used to hold local variables and function parameters?

When you pass a value to a function, the CPU pushes that value onto a *stack* in the computers memory.

```
void bar(int b)
{
  cout << b << endl;
}

void foo(int a)
{
  cout << a << endl;
  bar(a*2);
}

int main(void)
{
  int x = 5;
  foo( x );
}
```

Local variables are stored on the computer's built-in stack!

When you pass a value to a function, the CPU pushes that value onto a *stack* in the computers memory.

… when your function returns, the values are popped off the stack and go away.

Every time you *declare a local variable*, your program pushes it on the PC's stack automatically!

Output:

5
10

| | |
|---|---|
| b | 10 |
| a | 5 |
| x | 5 |

So how does the UNDO feature of your favorite word processor work?

It uses a stack, of course!

Every time you type a new word, it's added to the stack!

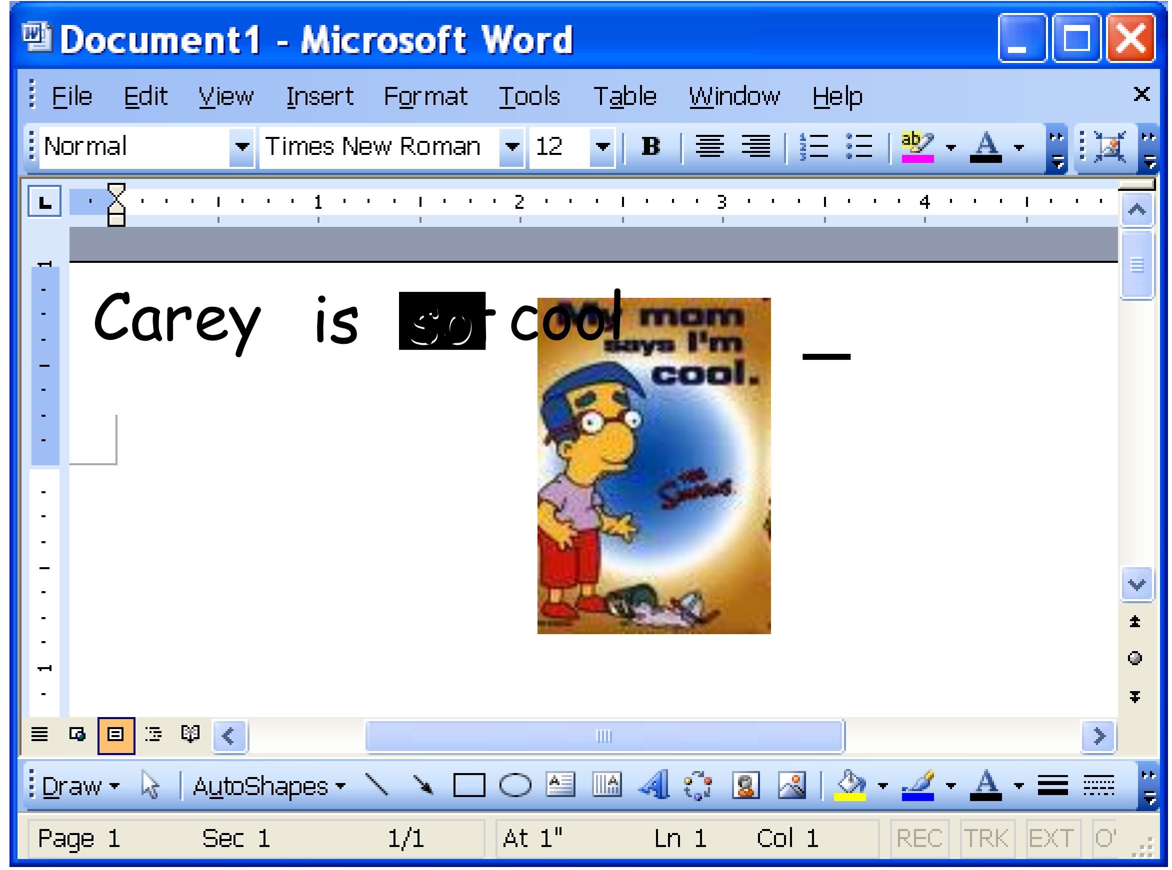
Every time you cut-and-paste an image into your doc, it's added to the stack!

And even when you delete text or pictures, this is tracked on a stack!

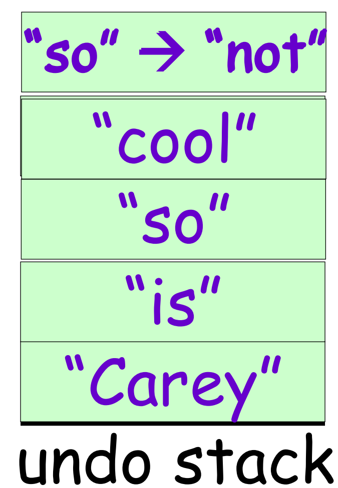
When the user hits the undo button…

The word processor pops the top item off the stack and removes it from the document!

In this way, the word processor can track the last X things that you did and properly undo them!







# Postfix Expression Evaluation

Most people are used to infix notation, where the operator is in-between the two operands, e.g.: A + B

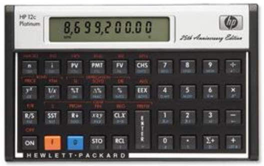
Postfix notation is another way to write algebraic expressions – here the operator follows the operands: A B +

Here are some infix expressions and their postfix equivalents:

| Infix | Postfix |
|---|---|
| 15 + 6 | 15 6 + |
| 9 – 4 | 9 4 - |
| (15 + 6) * 5 | 15 6 + 5 * |
| 7 * 6 + 5 | 7 6 * 5 + |
| 3 + (4 * 5) | 3 4 5 * + |

Postfix expressions are easier for a computer to compute than infix expressions, because they're *unambiguous*.

Ambiguous infix expression example: 5 + 10 * 3



# Postfix Evaluation Algorithm

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

$\boxed{7}$ 6 * 5 +

47

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

# Class Challenge

Given the following postfix expression: 6 8 2 / 3 * -

Show the contents of the stack after the 3 has been processed by our postfix evaluation algorithm.

Reminder:

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. If the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to the two #s (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

# Infix to Postfix Conversion

Stacks can also be used to convert infix expressions to postfix expressions:

For example,

From: (3 + 5) * (4 + 3 / 2) - 5
To: 3 5 + 4 3 2 / + * 5 -

Or

From: 3 + 6 * 7 * 8 - 3
To: 3 6 7 * 8 * + 3 -

Since people are more used to infix notation…

You can let the user type in an infix expression…

And then convert it into a postfix expression.

Finally, you can use the postfix evaluation alg (that we just learned) to compute the value of the expression.

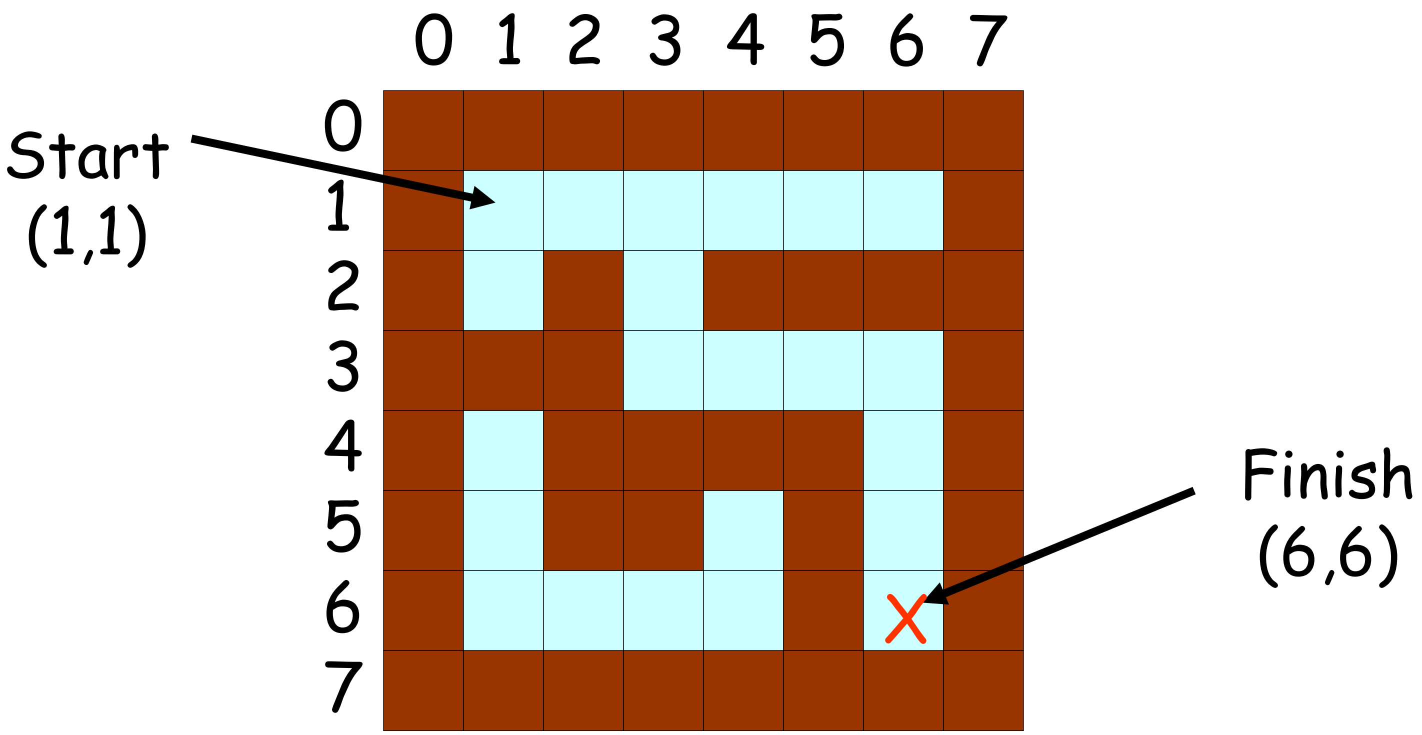
# Infix to Postfix Conversion

Inputs: Infix string
Output: postfix string (initially empty)
Private data: a stack

1. Begin at left-most Infix token.
2. If it's a #, append it to end of postfix string followed by a space
3. If its a "(", push it onto the stack.
4. If its an operator *and the stack is empty*:
   a. Push the operator on the stack.
5. If its an operator and the stack is NOT empty:
   a. Pop all operators with <u>greater or equal precedence</u> off the stack and append them on the postfix string.
   b. Stop when you reach an operator with lower precedence or a (.
   c. Push the new operator on the stack.
6. If you encounter a ")", pop operators off the stack and append them onto the postfix string until you pop a matching "(".
7. Advance to next token and GOTO #2
8. When all infix tokens are gone, pop each operator and append it } to the postfix string.

# Solving a Maze with a Stack!

We can also use a stack to determine if a maze is solvable:



# Solving a Maze with a Stack!

Inputs: 10x10 Maze in a 2D array,
Starting point (sx,sy)
Ending point (ex,ey)

Output: TRUE if the maze can be solved, FALSE otherwise

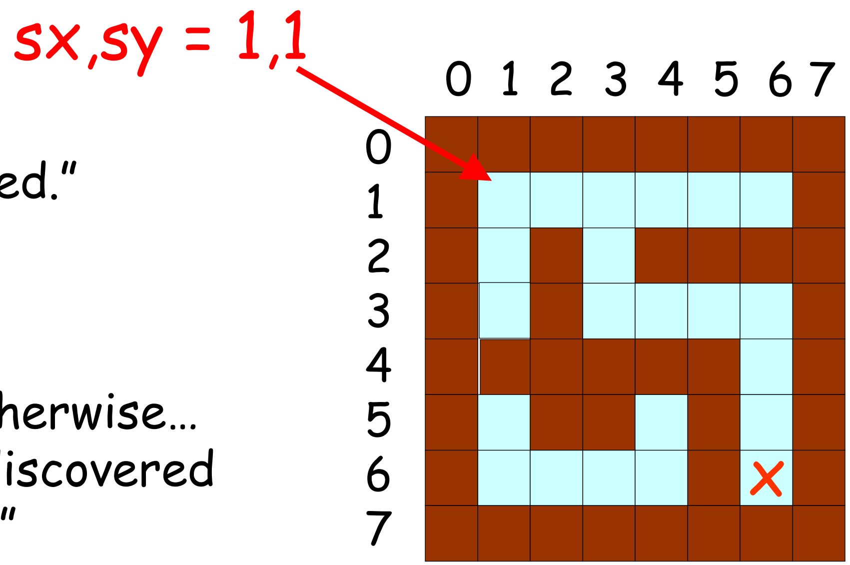
Private data: a stack of *points*

```
class Point
{
public:
  point(int x, int y);
  int getx() const;
  int gety() const;
private:
  int m_x, m_y;
};
```

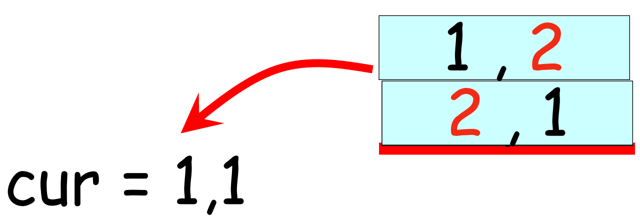
```
class Stack
{
public:
   Stack();     // c'tor
   void push(Point &p);
   Point pop();
   ...
private:
   ...
};
```

# Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. If the stack is empty, there is NO SOLUTION and we're done!
4. POP the top point off of the stack.
5. If we're at the endpoint, DONE! Otherwise…
6. If slot to the WEST is open & is undiscovered
   Mark (curx-1,cury) as "discovered"
   PUSH (curx-1,cury) on stack.
7. If slot to the EAST is open & is undiscovered
   Mark (curx+1,cury) as "discovered"
   PUSH (curx+1,cury) on stack.
8. If slot to the NORTH is open & is undiscovered
   Mark (curx,cury-1) as "discovered"
   PUSH (curx,cury-1) on stack.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   PUSH (curx,cury+1) on stack.
10. GOTO step #3

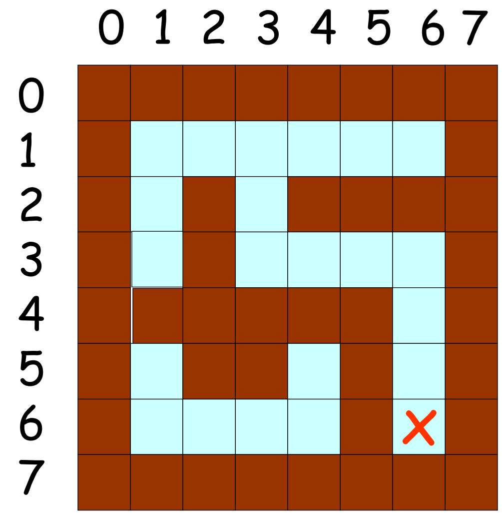



1,1 == 6,6?
Not yet!

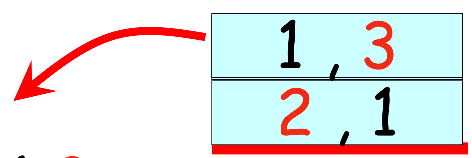


# Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. If the stack is empty, there is
   NO SOLUTION and we're done!
4. POP the top point off of the stack.
5. If we're at the endpoint, DONE! Otherwise…
6. If slot to the WEST is open & is undiscovered
   Mark (curx-1,cury) as "discovered"
   PUSH (curx-1,cury) on stack.
7. If slot to the EAST is open & is undiscovered
   Mark (curx+1,cury) as "discovered"
   PUSH (curx+1,cury) on stack.
8. If slot to the NORTH is open & is undiscovered
   Mark (curx,cury-1) as "discovered"
   PUSH (curx,cury-1) on stack.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   PUSH (curx,cury+1) on stack.
10. GOTO step #3



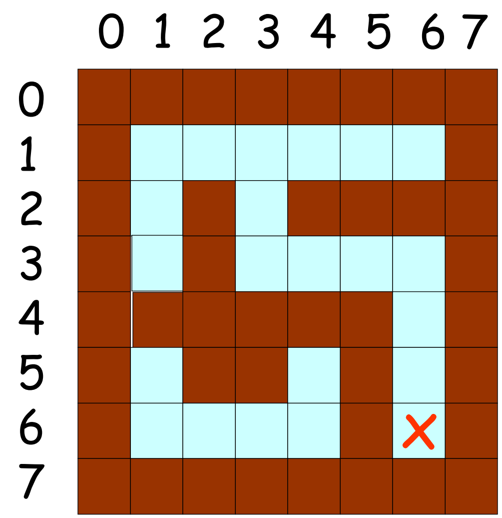

1,2 == 6,6?
Not yet!



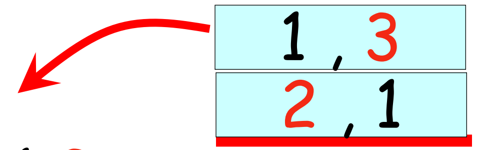

cur = 1,2

# Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. If the stack is empty, there is NO SOLUTION and we're done!
4. POP the top point off of the stack.
5. If we're at the endpoint, DONE! Otherwise…
6. If slot to the WEST is open & is undiscovered
   Mark (curx-1,cury) as "discovered"
   PUSH (curx-1,cury) on stack.
7. If slot to the EAST is open & is undiscovered
   Mark (curx+1,cury) as "discovered"
   PUSH (curx+1,cury) on stack.
8. If slot to the NORTH is open & is undiscovered
   Mark (curx,cury-1) as "discovered"
   PUSH (curx,cury-1) on stack.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   PUSH (curx,cury+1) on stack.
10. GOTO step #3



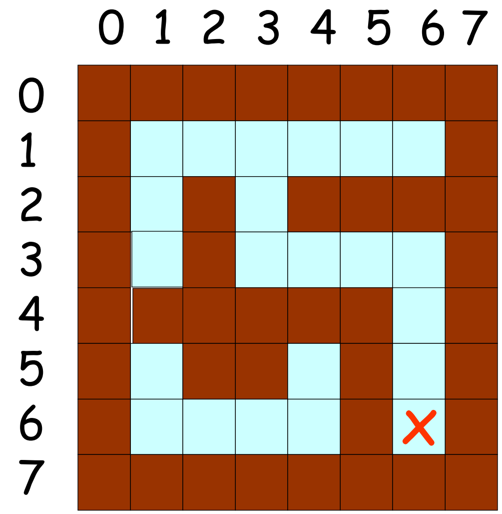

1,3 == 6,6?
Not yet!




cur = 1,3

# Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. If the stack is empty, there is NO SOLUTION and we're done!
4. POP the top point off of the stack.
5. If we're at the endpoint, DONE! Otherwise…
6. If slot to the WEST is open & is undiscovered
   Mark (curx-1,cury) as "discovered"
   PUSH (curx-1,cury) on stack.
7. If slot to the EAST is open & is undiscovered
   Mark (curx+1,cury) as "discovered"
   PUSH (curx+1,cury) on stack.
8. If slot to the NORTH is open & is undiscovered
   Mark (curx,cury-1) as "discovered"
   PUSH (curx,cury-1) on stack.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   PUSH (curx,cury+1) on stack.
10. GOTO step #3



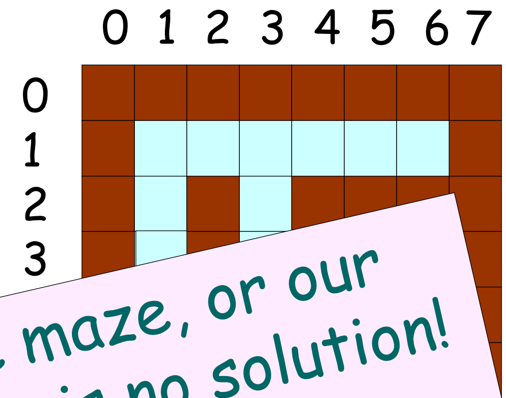

2,1 == 6,6?
Not yet!

3 , 1

cur = 2,1

# Solving a Maze with a Stack!

1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. If the stack is empty, there is
   NO SOLUTION and we're done!
4. POP the top point off of the stack.
5. If we're at the endpoint, DONE! Otherw
6. If slot to the WEST is open & i
   Mark (curx-1,cu
   PUS
8. open & is undiscovered
   ,cury-1) as "discovered"
   PUSH (curx,cury-1) on stack.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   PUSH (curx,cury+1) on stack.
10. GOTO step #3




,1 == 6,6?
Not yet!

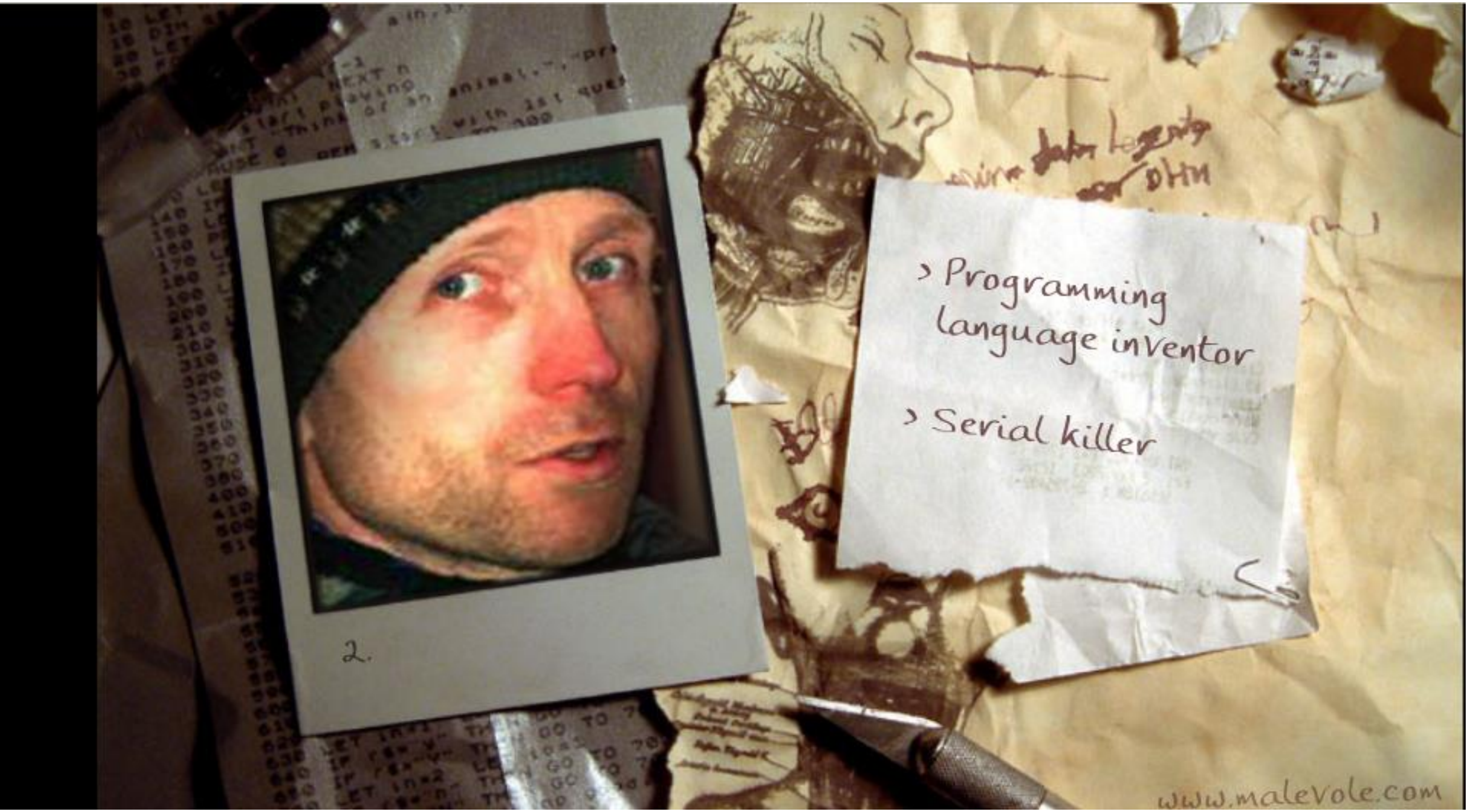
Eventually, we'll find the solution to the maze, or our stack will empty out, indicating that there is no solution!

This searching algorithm is called a "depth-first search."

| |
|---|
| 3 , 2 |
| 4 , 1 |

cur = 3,1

# Your favorite game!



# The Queue

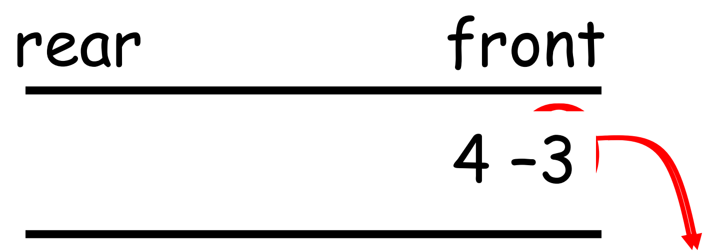
# Another ADT: The Queue

The queue is another ADT that is just a like a line at the store or at the bank.

The first person in line is the first person out of line and served.

This is called a FIFO data structure:
FIRST IN, FIRST OUT.

Every queue has a *front* and a *rear*. You enqueue items at the *rear* and dequeue from the front.




What data structures could you use to implement a queue?

# The Queue Interface

enqueue(int a):
Inserts an item on the rear of the queue

int dequeue():
Removes and returns the top item from the front of the queue

bool isEmpty():
Determines if the queue is empty

int size():
Determines the # of items in the queu

int getFront():
Gives the value of the top item on the
without removing it like dequeue

Like a Stack, we can have queues of any type of data! Queues of strings, Points, Nerds, ints, etc!

# Common Uses for Queues

Often, data flows from the Internet faster than the computer can use it. We use a queue to hold the data until the browser is ready to display it...

Every time your computer receives a character, it enqueues it:

```
internetQueue.enqueue(c);
```

Every time your Internet browser is ready to get and display new data, it looks in the queue:

```
while (internetQueue.isEmpty() == false)
{
    char ch = internetQueue.dequeue();

    cout << ch;  // display web page...
}
```

# Common Uses for Queues

You can also use queues to search through mazes!

If you use a queue instead of a stack in our searching algorithm, it will search the maze in a different order…

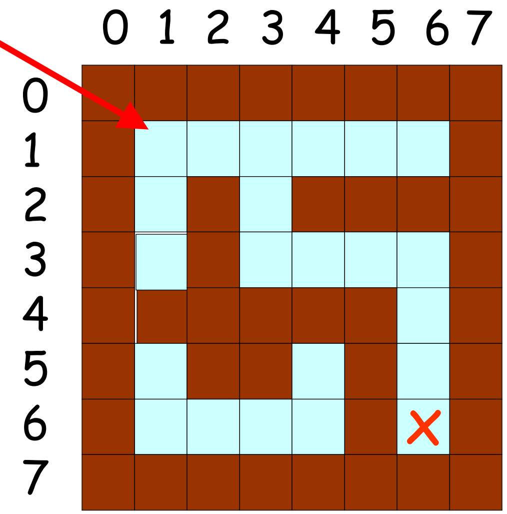
Instead of always exploring the last x,y location pushed on top of the stack first…

The new algorithm explores the oldest x,y location inserted into the queue first.

# Solving a Maze with a Queue!

(AKA Breadth-first Search)

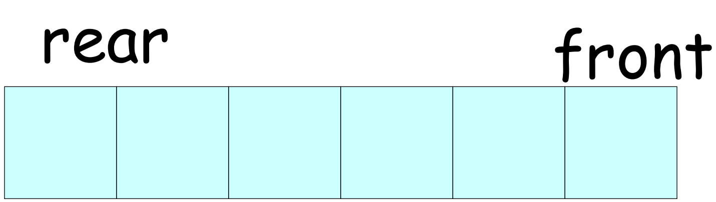
sx,sy = 1,1




1. Insert starting point onto the queue.
2. Mark the starting point as "discovered."
3. If the queue is empty, there is NO SOLUTION and we're done!
4. Remove the top point from the queue.
5. If we're at the endpoint, DONE! Otherwise…
6. If slot to the WEST is open & is undiscovered
   Mark (curx-1,cury) as "discovered"
   INSERT (curx-1,cury) on queue.
7. If slot to the EAST is open & is undiscovered
   Mark (curx+1,cury) as "discovered"
   INSERT (curx+1,cury) on queue.
8. If slot to the NORTH is open & is undiscovered
   Mark (curx,cury-1) as "discovered"
   INSERT (curx,cury-1) on queue.
9. If slot to the SOUTH is open & is undiscovered
   Mark (curx,cury+1) as "discovered"
   INSERT (curx,cury+1) on queue.
10. GOTO step #3

And so on…

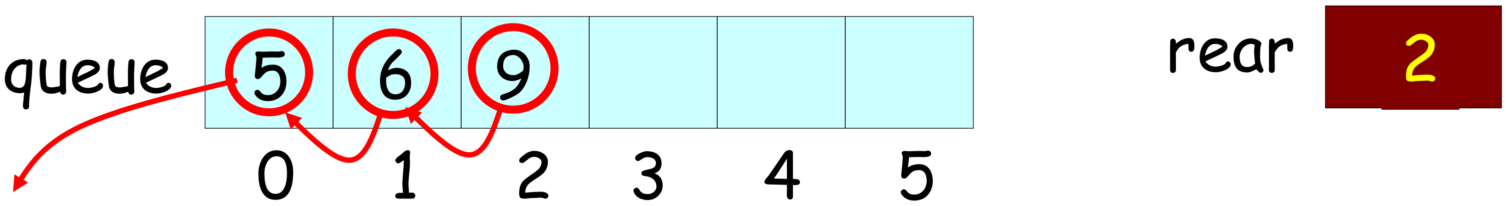
curx,cury=



# Queue Implementations

We can use an array and an integer to represent a queue:

```
int queue[6], rear = 0;
```




- Every time you insert an item, place it in the rear slot of the array and increment the rear count
- Every time you dequeue an item, move all of the items forward in the array and decrement the rear count.

What's the problem with the array-based implementation?

If we have N items in the queue, what is the cost of:
(1) inserting a new item, (2) dequeuing an item

# Queue Implementations

We can also use a linked list to represent a queue:

- Every time you insert an item, add a new node to the end of the linked list.
- Every time you dequeue an item, take it from the head of the linked list and then delete the head node.

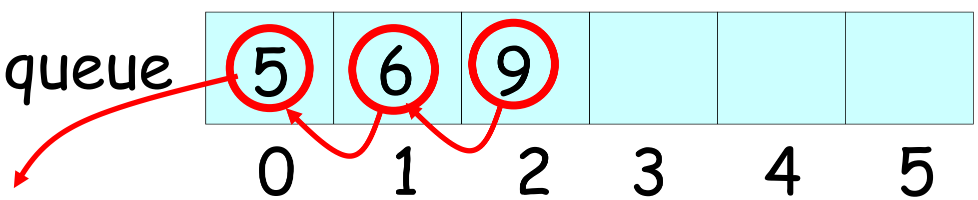
Of course, you'll want to make sure you have both head and tail pointers…

or your linked-list based queue will be really inefficient!

# The Circular Queue

The circular queue is a clever type of array-based queue.

Unlike our previous array-based queue, we never need to shift items with the circular queue!




Let's see how it works!

# The Circular Que

If the count is zero, then you know the queue is empty. If the count is N, you know it's full…

Private data:

an array: arr
an integer: head
an integer: tail
an integer: count

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 4 | -1 | 9 | 7 | 5 |

tail 1

head 3

count 4

- To initialize your queue, set:
  count = head = tail = 0
- To insert a new item, place it in arr[tail] and then increment the tail & count values
- To dequeue the head item, fetch arr[head] and increment head and decrement count
- If the head or tail go past the end of the array, set it back to 0.

Enqueue: 6
Enqueue: 4
Enqueue: -1
Dequeue -> 6
Enqueue: 9
Enqueue: 7
Dequeue -> 4
Enqueue: 5
Enqueue: 42
Dequeue -> -1

# A Queue in the STL!

The people who wrote the Standard Template Library also built a queue class for you:

```
#include <iostream>
#include <queue>

int main()
{
      std::queue<int>  iqueue;   // queue of ints

      iqueue.push(10);           // add item to rear
      iqueue.push(20);
      cout << iqueue.front();    // view front item
      iqueue.pop();              // discard front item
      if (iqueue.empty() == false)
        cout << iqueue.size();
}
```

# Class Challenge

Given a circular queue of 6 elements, show the queue's contents, and the Head and Tail pointers after the following operations are complete:

enqueue(5)
enqueue(10)
enqueue(12)
dequeue()
enqueue(7)
dequeue()
enqueue(9)
enqueue(12)
enqueue(13)
dequeue()