CHAPTER 3
STACKS AND QUEUES

Iris Hui-Ru Jiang

Stacks and Queues

- **Contents**
  - Templates in C++
  - Stack (LIFO)
  - Queue (FIFO)
  - Subtyping and Inheritance in C++
  - A Mazing Problem
  - Evaluation of Expressions
- **Readings**
  - Chapter 3
  - C++ STL
    - stack
    - queue
    - deque

How to Reuse Implemented Functions?

SelectionSort on ints

```
void SelectionSort (int *a, const int n)
{// Sort the n integers a[0] to a[n-1] into
      nondecreasing order
   for (int i = 0; i < n; i++)
   {
      int j = i;
      // find smallest integer in a[i] to a[n-1]
      for (int k = i + 1; k < n; k++)
         if (a[k] < a[j]) j = k;
      swap(a[i], a[j]);
   }
}
```

SelectionSort on floats

```
void SelectionSort (float *a, const int n)
{// Sort the n floating points a[0] to a[n-1] into
      nondecreasing order
   for (int i = 0; i < n; i++)
   {
      int j = i;
      // find smallest integer in a[i] to a[n-1]
      for (int k = i + 1; k < n; k++)
         if (a[k] < a[j]) j = k;
      swap(a[i], a[j]);
   }
}
```

What if we wish to sort an array of **float**s instead of **int**s?
1. Replace **int** with **float** using a text editor – tedious!
2. Better idea?

C++ Templates

SelectionSort on Templates

```
template <class T>
void SelectionSort (T *a, const int n)
{// Sort the n integers a[0] to a[n-1] into
      nondecreasing order
   for (int i = 0; i < n; i++)
   {
      int j = i;
      // find smallest integer in a[i] to a[n-1]
      for (int k = i + 1; k < n; k++)
         if (a[k] < a[j]) j = k;
      swap(a[i], a[j]);
   }
}
```

```
float farray[100];
int intarray[250];
.
.
SelectionSort(farray, 100);
SelectionSort(intarray, 250);
```

instantiation

Templates

- **A template: a parameterized data type, it can be**
  1. Fundamental C++ type
  2. User-defined type
- **Sort *Rectangles*?**
  - Overload **operator**<

Representing Container Classes (1/2)

Bag of ints

```
class Bag {
public:
  Bag(int bagCapacity = 10);  // constructor
  ~Bag(); // destructor

  int Size() const; // return # of elements in bag
  bool IsEmpty() const; // is the bag empty?
  int Element() const; // return an element inside
  void Push(const int); // insert an integer
  void Pop(); //delete an integer

private:
  int *array;
  int capacity; // capacity of array
  int top; // array position of top element
};
```

Operations of *Bag*

```
Bag::Bag(int bagCapacity):
  capacity (bagCapacity) {
  if (capacity < 1) throw “Capacity must be > 0”;
  array = new int[capacity]; top = -1; }
Bag::~Bag() { delete [] array; }

inline int Bag::Size() const { return top+1; }
inline bool Bag::IsEmpty() { return Size() == 0;}
inline int Bag::Element() const {
  if (IsEmpty()) throw “Bag is empty”;
  return array[0]; }

void Bag::Push(const int x) {
  if (capacity == top+1) { ChangeSize1D(array,
      capacity, 2*capacity); capacity *=2; }
  array[++top]=x; }
void Bag::Pop() {
  if (IsEmpty()) throw “Bag empty! cannot delete”;
  int deletePos = top/2;
  copy(array+deletePos+1, array+top+1,
      array+deletePos); // compact array
  top--; }
```

Representing Container Classes (2/2)

Template Class *Bag*

```
template <class T>
class Bag {
public:
  Bag(int bagCapacity = 10);
  ~Bag();

  int Size() const;
  bool IsEmpty() const;
  T& Element() const;
  void Push(const T&);
  void Pop();

private:
  T *array;
  int capacity; // capacity of array
  int top; // array position of top element
};
```

instantiation

```
Bag<int> a;
Bag<Rectangle> r;
```

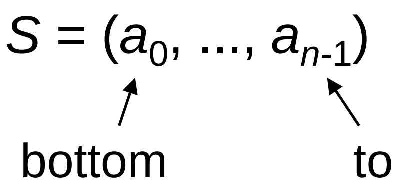
Operations of *Bag*

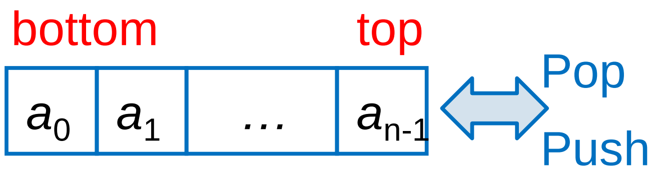
```
template <class T>
Bag<T>::Bag(int bagCapacity):
  capacity (bagCapacity) {
  if (capacity < 1) throw "Capacity must be > 0";
  array = new T[capacity]; top = -1; }
template <class T>
Bag<T>::~Bag() { delete [] array; }

template <class T>
void Bag<T>::Push(const T& x) {
  if (capacity == top+1) { ChangeSize1D(array,
      capacity, 2*capacity); capacity *=2; }
  array[++top]=x; }
template <class T>
void Bag<T>::Pop() {
  if (IsEmpty()) throw "Bag empty! cannot delete";
  int deletePos = top/2;
  copy(array+deletePos+1, array+top+1,
      array+deletePos); // compact array
  array[top--].~T(); }
```

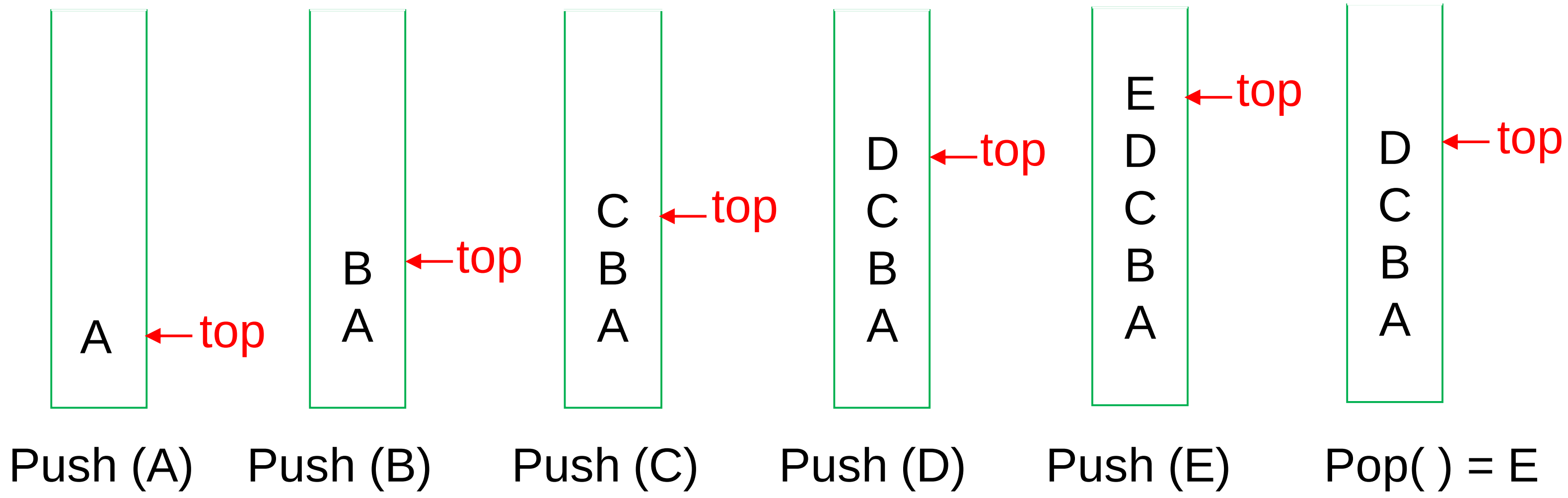
ADT Stack

What is a Stack?

- **Definition: A stack is**
  - An ordered list in which insertions (Push) and deletions (Pop) are made at one end, called the top
  - Also called a Last-In-First-Out (LIFO) list
  - Application: system stack used to process function calls, etc.



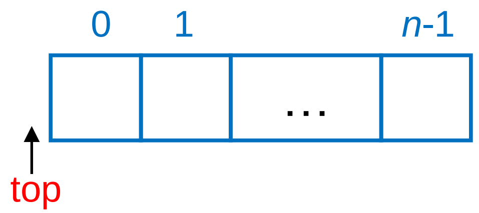




ADT *Stack*

```
template <class T>
class Stack
{ // objects: A finite ordered list with zero or more elements
public:
    Stack (int stackCapacity = 10);
    // create an empty stack of initial capacity is stackCapacity

    ~Stack () {delete [] stack;}
    // destroy the stack

    bool IsEmpty() const;
    // if # of elements in the stack is 0, return true else return false

    T& Top() const;
    // return top element of stack

    void Push (const T& item);
    // insert item into top of stack

    void Pop();
    // delete top element
};
```

Stack Implementation: 1D Array



```
template <class T>
class Stack {
public:
      Stack (int stackCapacity = 10);  // ctor
      ~Stack(); // dtor
      bool IsEmpty();
      T& Top() const;
      void Push (const T& item);
      void Pop();
private:
      T *stack; // stack array
      int top; // array position of top
      int capacity; // capacity of stack array
};
```

1D array

```
template <class T>
Stack<T>::Stack(int stackCapacity)
  :capacity (stackCapacity) {
      if (capacity < 1) throw
          "Stack capacity must be > 0";
      stack = new T[capacity]; top = -1; }

template <class T>
Stack<T>::~Stack() { delete [] stack; }
```

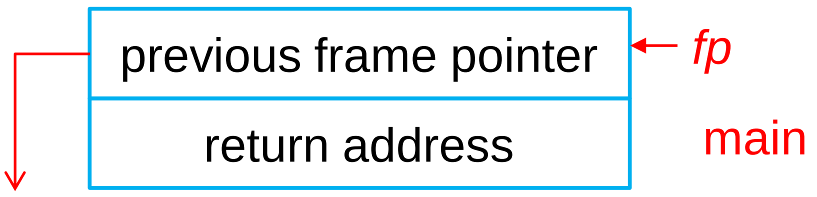
```
template <class T>
inline bool Stack<T>::IsEmpty() const {
      return top == – 1;}

template <class T>
inline T& Stack<T>::Top() const {
      if (IsEmpty()) throw "Stack empty!";
      return stack[top]; }

template <class T>
void Stack<T>::Push(const T& x) {
      if (top==capacity-1) {
          ChangeSize1D(stack, capacity,
          2*capacity);
          capacity *=2; }
      stack[++top] = x; // insert from top
}

template <class T>
void Stack<T>::Pop() {
      if (IsEmpty())
          throw "Stack empty! Cannot delete";
      stack[top--].~T(); // delete from top
}
```

Application: System Stack

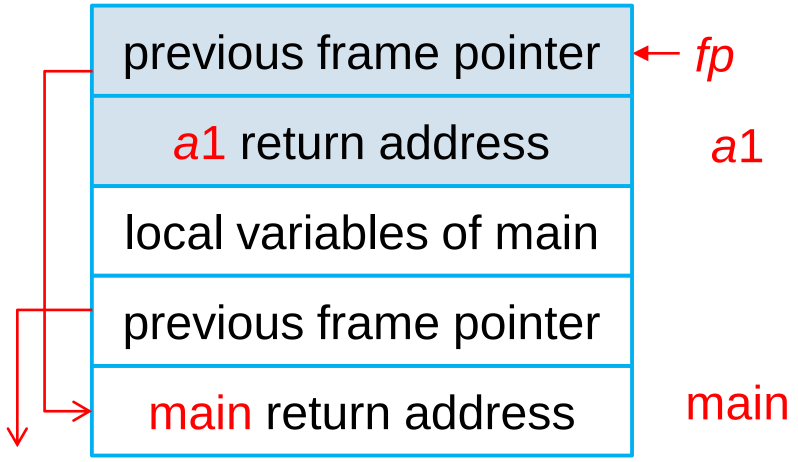
- **The system stack is used at runtime to process function calls**

fp: a pointer to current stack frame

stack frame of invoking function



System Stack before *a*1 is invoked

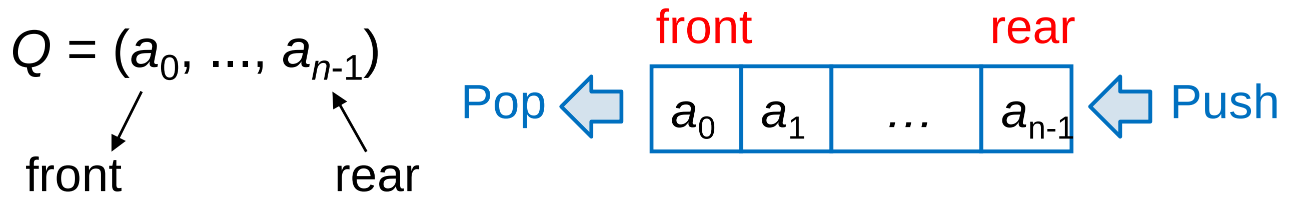


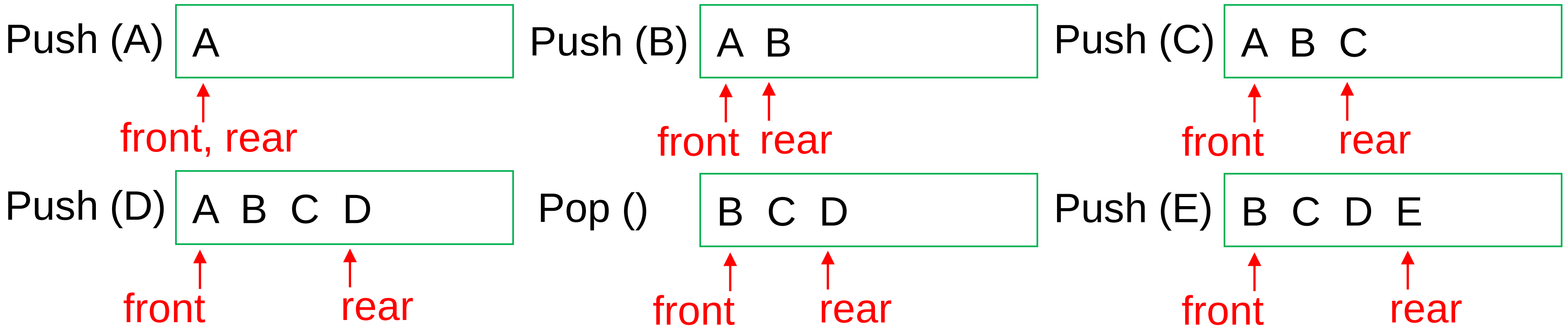
System Stack after *a*1 is invoked

ADT Queue

What is a Queue?

- **Definition: A queue is**
  - An ordered list in which insertions (Push) take place at one end, called rear, and deletions (Pop) take place at the other end, called front
  - Also called a First-In-First-Out (FIFO) list





ADT *Queue*

template <**class** *T*>
class *Queue*
{ // objects: A finite ordered list with zero or more elements
public:
    Queue (**int** *queueCapacity* = 10);
    // create an empty queue of initial capacity *queueCapacity*
    ~*Queue* () {**delete** [] *Queue*;}
    // destructor

    bool *IsEmpty*();
    // if # of elements in the queue is 0, return **true** else return **false**

    T& *Front*() **const;**
    // return front element

    T& *Rear*() **const;**
    // return rear element

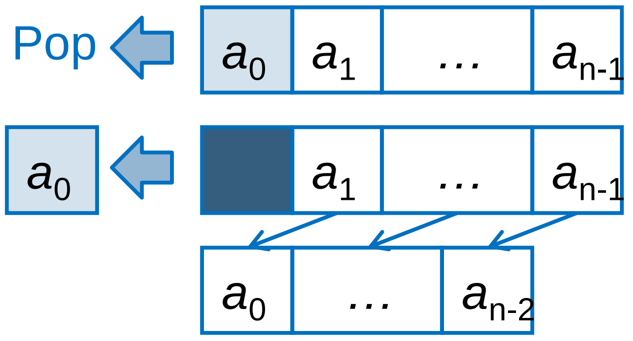
    void *Push* (**const** *T*& *item*);
    // insert *item* at the rear of the queue

    void *Pop*();
    // delete the front element

};

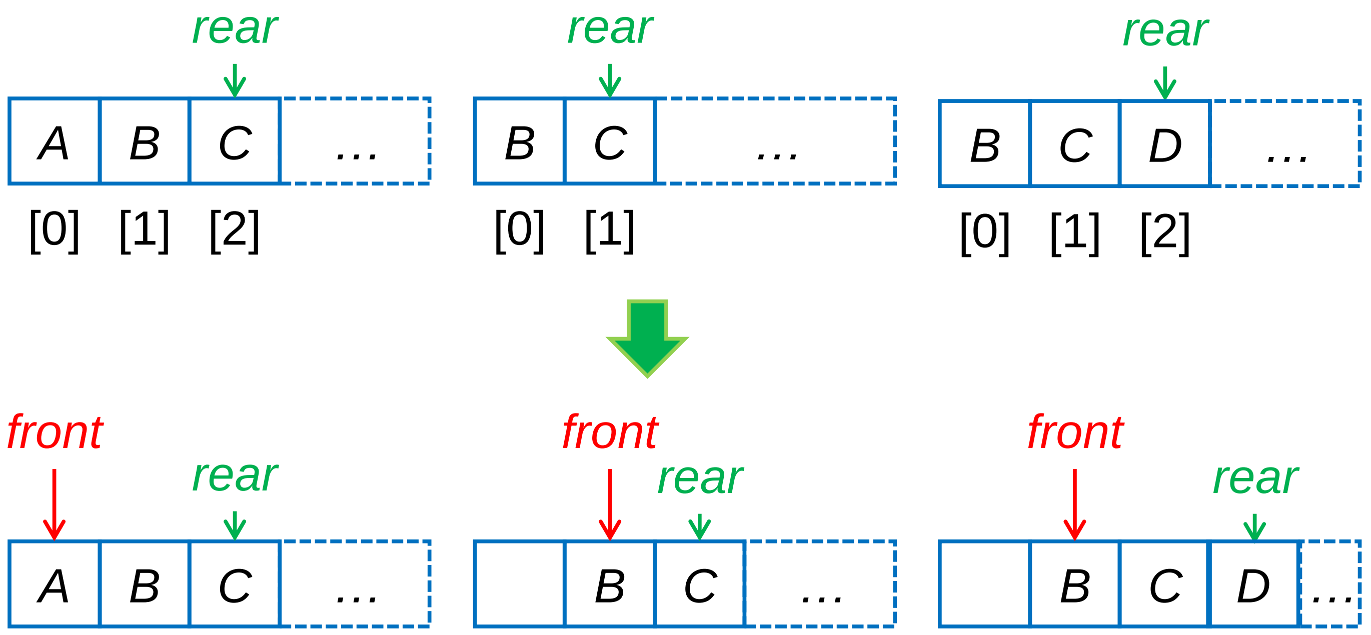
$\Theta(1)$:
If no resizing

$\Theta(n)$:
remove&shift



How to Speed up Pop?

- **Pop in Θ(1)?**
  - Remedy: Relax the requirement front at a_0
    - Use two pointers: *front*, *rear*
      - *front* points to the front element
      - *rear* points to the rear element



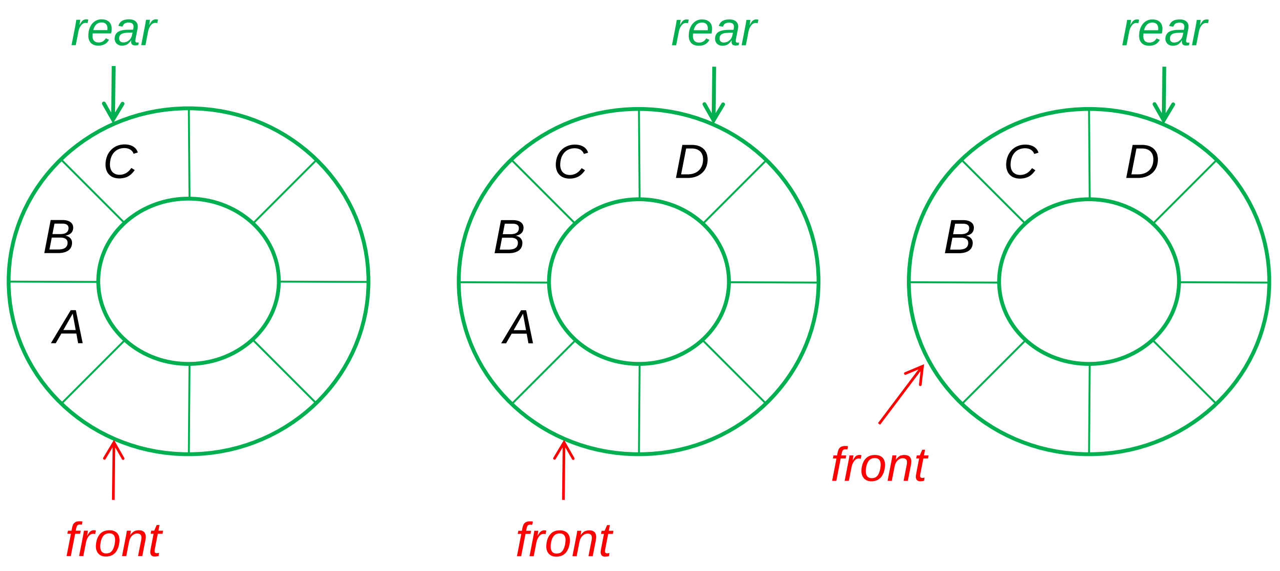
Circular Queue

- **What if pushing when *rear == capacity* -1 and *front* > 0?**
  - Remedy: circular queue

```
if (rear == capacity -1) rear =0;
else rear++;
```

→

```
(rear+1)%capacity;
```



front @ *front+1*
rear @ *rear*

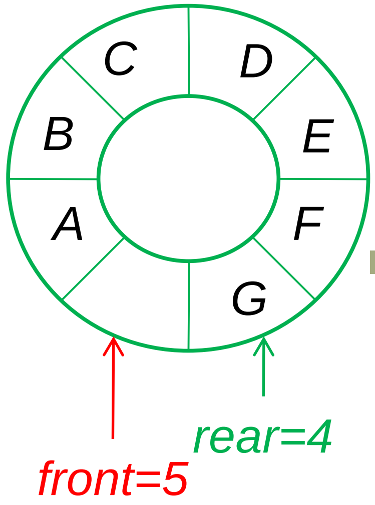
- **Cannot distinguish empty: *front == rear* from full: *front == rear***
  - Remedy: enlarge *capacity* just before queue full

Circular Queue Implementation

```
template <class T>
class Queue {
public:
   Queue(int queueCapacity = 10);
   ~Queue();
   bool IsEmpty();
   T& Front() const;
   T& Rear() const;
   void  Push (const T& item);
   void  Pop();
private:
   T *queue;
   int front,  // one counterclockwise from front
       rear, capacity;
};

template <class T>
Queue<T>::Queue(int queueCapacity):
   capacity( queueCapacity) {
   if (capacity<1) throw "capacity must be >0";
   queue = new T[capacity];
   front = rear = 0;}
template <class T>
Queue<T>::~Queue() {delete [] queue;}
```

```
template <class T>
inline bool Queue<T>::IsEmpty() {
   return front == rear;}
template <class T>
inline T& Queue<T>::Front() {
   if (IsEmpty()) throw "Queue empty!";
   return queue[(front+1)%capacity];}
template <class T>
inline T& Queue<T>::Rear() {
   if (IsEmpty()) throw "Queue empty!";
   return queue[rear];}

template <class T>
void Queue<T>::Push(const  T& x) {
   if ((rear+1)%capacity==front) {
      // double capacity right before full
         // code to double capacity comes here}
   rear=(rear+1)%capacity; queue[rear] = x;}

template <class T>
void Queue<T>::Pop() {
   if (IsEmpty()) throw "Queue empty! No pop";
   front = (front+1)%capacity;
   queue[front].~T();}
```

Doubling Queue Capacity in Flattened View



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	*D*	*E*	*F*	*G*		*A*	*B*

front=5, *rear=4*

Data encapsulation: Choose one of them without modifying codes

- **Configuration 1: double & slide**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	*D*	*E*	*F*	*G*		*A*	*B*								

front=5, *rear=4*

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	*D*	*E*	*F*	*G*										*A*	*B*

front=13, *rear=4*

- **Configuration 2: relocate**

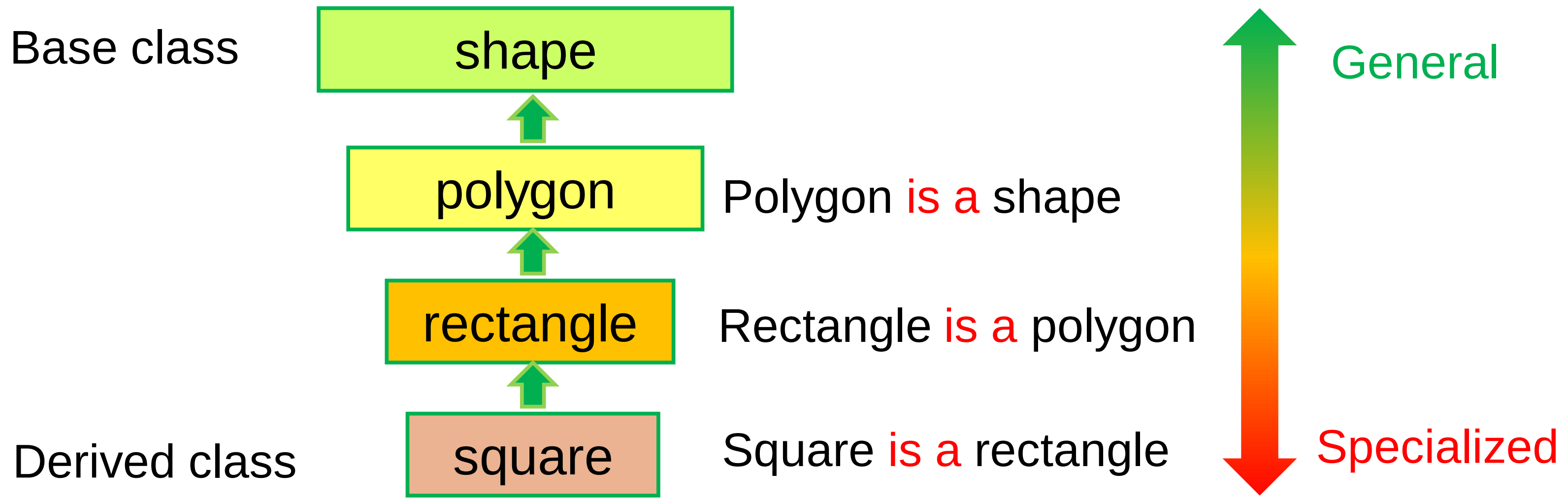
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	*B*	*C*	*D*	*E*	*F*	*G*									

front=15, *rear=6*

Subtype & Inheritance in C++

Public Inherence

- **Is used to express the IS-A relationship**
  - If B inherits from A, then B IS-A A. A is more general than B.



- **A derived class**
  - inherits all the non-private members (data and functions) of the base class (except constructor and destructor)
  - can override the inherited functions for its own needs

A Stack IS A Bag (1/2)

```
class Bag {
    public:
        Bag (int bagCapacity = 10);
        virtual ~Bag();

        virtual int Size() const;
        virtual bool IsEmpty() const;
        virtual int Element() const;

        virtual void Push(const int);
        virtual void Pop();
    protected:
        int *array;
        int capacity;
        int top;
};

class Stack: public Bag {
    public:
        Stack(int stackCapacity = 10);
        ~Stack();
        int Top() const;
        void Pop();
};
```

Stack can reuse non-private members of *Bag*

A Stack IS A Bag (2/2)

```
Stack:: Stack (int stackCapacity) : Bag(stackCapacity) { }
// Constructor for Stack calls constructor for Bag

Stack::~Stack() { }
// Destructor for Bag is automatically called when Stack is destroyed.
// This ensures that array is deleted.

int Stack::Top() const
{
    if (IsEmpty()) throw "Stack is empty.";
    return array[top];
}

void Stack::Pop()
// Bag::Pop is different from Stack::Pop
// => taylor one
{
    if (IsEmpty()) throw "Stack is empty. Cannot delete.";
    top--;
}
```

Redefine operations:
ctor, dtor, *Top*(), *Pop*()

Example:

```
Stack s(3); // uses Stack ctor to create array of size 3
s.Push(1); s.Push(2); s.Push(3);
// Stack::Push not defined, so use Bag::Push
s.Pop();
// uses Stack::Pop, which calls Bag::IsEmtpy
// because IsEmpty has not been redefined in Stack

s.Size(); // uses Bag::Size
s.Element(); // uses Bag::Element
```

A Mazing Problem

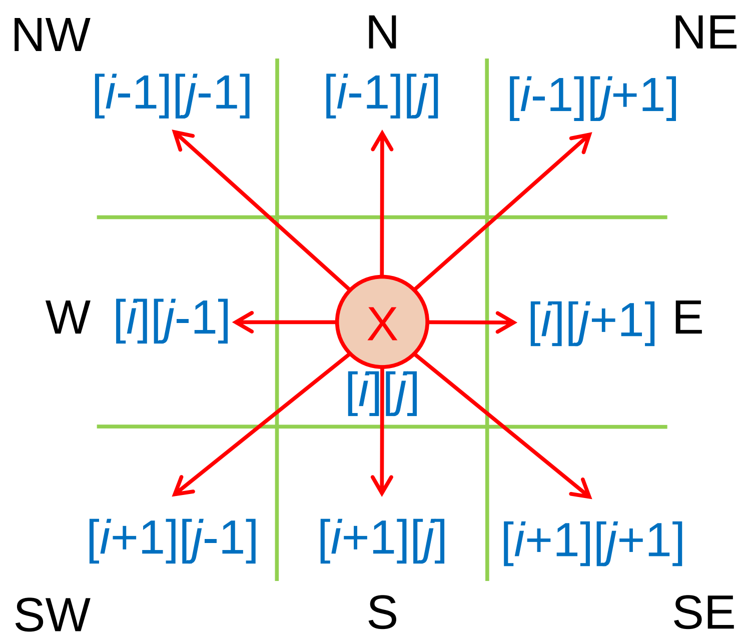
Stack Application

Rat-in-a-Maze

- **A maze is represented by:** *maze*[1..*m*][1..*p*]
  - 0: through path; 1: blocked path
  - Can you find a path?
- **Q: How to implement the wall (boundary)?**
  - *maze*[0..*m*+1][0..*p*+1]

Entrance →	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	
	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	
	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	
	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1	
	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	
	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	
	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	→ Exit

Making A Move



q	*move*[*q*].*a*	*move*[*q*].*b*
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

```
struct offsets { int a, b; }
enum directions { N, NE, E, SE, S, SW, W, NW };
offsets move[8];
```

- **Move from [*i*][*j*] southwest to [*g*][*h*]**
  - *g* = *i* + *move*[*SW*].*a*; *h* = *j* + *move*[*SW*].*b*
  - What if [*i*][*j*] is at the border?

How to Find a Path?

- **At each location, we examine all possibilities, start from the north and look clockwise**
- **We may need to trace back…**
  - Save the current position and the direction of the last move in a list (stack)
- **We do not repeat the same path…**
  - Use of another array *mark*[1..*m*][1..*p*] to mark visited positions
    - 0: initial; 1: visited
- **We may need to know the whole path…**
  - Retrieve it from the stack

Path Finding Algorithm

```
initialize list to the maze entrance coordinates and direction east;
while (list is not empty)
{
    (i, j, dir) = coordinates and direction from end of list;
    delete last element of list;
    while (there are more moves from (i, j))
    {
        (g, h) = coordinates of next move;
        if ((g == m) && (h == p)) success;
        if ((!maze[g][h] && (!mark[g][h])) // legal and unvisited
        {
            mark[g][h] = 1;
            dir = next direction to try;
            add (i, j, dir) to end of list;
            (i, j, dir) = (g, h, N);
        }
    }
}
cout << “No path in maze.” << endl;
```

Evaluation of Expressions

Stack Application

How to Evaluate an Expression?

- **Example:** $X = A / B - C + D * E - A * C$
  - Let $A = 4$, $B = C = 2$, $D = E = 3$
  - Evaluation 1: ((4/2)–2)+(3*3)–(4*2) = 0 + 9 – 8 = 1
  - Evaluation 2: (4/(2–2+3))*(3–4)*2 = (4/3)*(–1)*2 = -2.666…
- **In what order to carry out the answer in C/C++?**
  - Proceed operators of the same priority left to right
  - Use parentheses to override the rules, e.g., $A / (B - C)$
    - $X = ((((A / B) - C) + (D * E)) - (A * C))$

priority	operator
1	unary minus, !
2	*, /, %
3	+, –
4	<. <=, >, >=
5	==, !=
6	&&
7	||

Infix vs. Postfix

a+b	ab+	+ab
infix	postfix	prefix

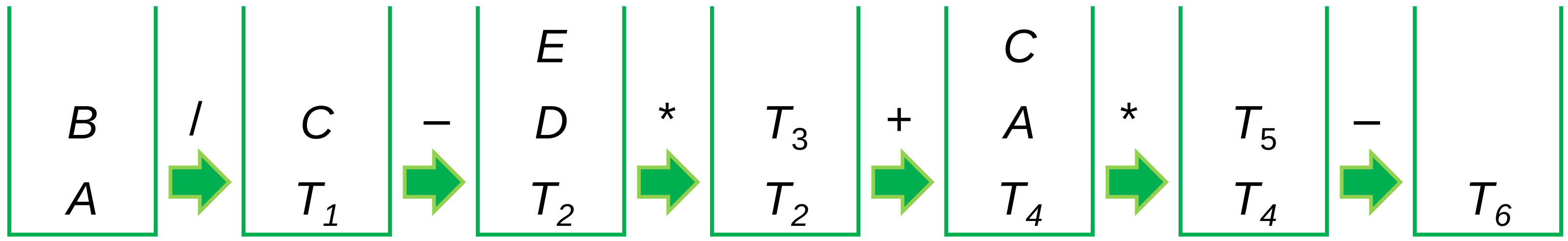
- **Infix: the operator in-between operands**
- **Postfix: the operator after operands**
- **Prefix: the operator before operands**
- *A* / *B* – *C* + *D* * *E* – *A* * *C*
  - Parenthesized: ((((*A* / *B*) – *C*) + (*D* * *E*)) – (*A* * *C*))
  - Postfix: *A B* / *C* – *D E* *+ *A C* * –
- **Why postfix?**
  - Parenthesis free (priority is no longer relevant)
  - Evaluation done by making only one left to right scan

Evaluating a Postfix Expression (1/2)

- **Make a left to right scan**
- **Stack operands**
- **Evaluate operators using operands from the stack**
- **Place the result onto the stack**

- **Example:** $A / B - C + D * E - A * C$
  - Postfix: $A\ B\ /\ C - D\ E\ * + A\ C\ * -$

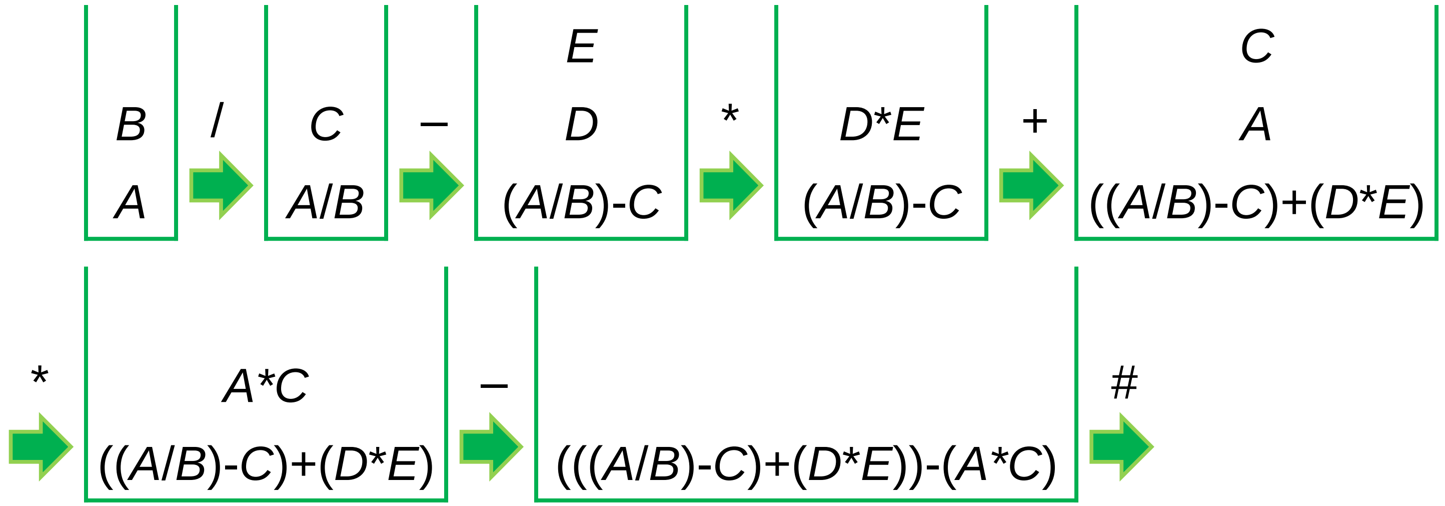
operation	postfix
$T_1 = A / B$	$T_1C-DE*+AC*-$
$T_2 = T_1 - C$	$T_2DE*+AC*-$
$T_3 = D * E$	T_2T_3+AC*-
$T_4 = T_2 + T_3$	T_4AC*-
$T_5 = A * C$	T_4T_5-
$T_6 = T_4 - T_5$	T_6



Evaluating a Postfix Expression (2/2)

```
void Eval (Expression e) {
// Evaluate the postfix expression e.
// Assume that the last token in e is '#.' (a token is either an operator, operand, or '#.')
// NextToken() gets the next token from e.
// Eval() uses the stack stack
    Stack<Token> stack; // initialize stack
    for (Token x = NextToken(e); x != '#'; x=NextToken(e))
        if (x is an operand) stack.Push(x); // add to stack
        else { // operator
            remove the correct number of operands for operator x from stack;
            perform the operation x and store the result (if any) onto the stack;
        }
}
```

e = *A B* / *C* - *D E* *+ *A C* * – #



Infix to Postfix – Method 1

- **Fully parenthesize the expression**
- **Move all operators so that they replace their corresponding right parentheses**
- **Delete all parentheses**

- **Example:** *A / B – C + D * E – A * C*
  - $$\begin{tikzcd}[column sep=tiny] (& (& (& (& A & / \arrow[rr, bend right] & B &) & - \arrow[rr, bend right] & C &) & + \arrow[rrrrrr, bend right] & (& D & * \arrow[rr, bend right] & E &) &) & - \arrow[rrrrrr, bend right] & (& A & * \arrow[rr, bend right] & C &) &) \end{tikzcd}$$
  - *((((A   B /   C –   (D   E * +   (A   C * –*
  - *A B / C – D E *+ A C * –*

Infix to Postfix – Method 2 (1/4)

- **Observation: operands**
  - Have the same order in both infix and postfix
  - $\Rightarrow$ Pass operands immediately to output
- **Observation: operators**
  - *A * B – C* *AB*C–*
  - *A + B * C* *ABC*+*
  - $\Rightarrow$ Hold operators for a while and pass them out at the right time
    - Check priority
- **Observation: parentheses**
  - *A * (B + C) / D* *ABC+*D/*

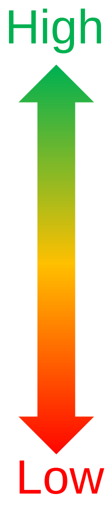
Infix to Postfix – Method 2 (2/4)

- **Rules:**
  1. Scan the infix expression from left to right
  2. If the token is an operand, output it immediately
  3. If the token is an operator
     - If its priority is higher than that of the top (of the stack)
       - Push the operator into the stack
     - If its priority is equal or lower than that of the top
       - Unstack until its priority is higher than that of the new top
       - Push the operator into the stack
  4. If the token is '('
     - Push '(' and following operators into stack until ')'
     - Unstack until '('
  5. If the token is '#', empty stack

Infix to Postfix – Method 2 (3/4)

Example: $e = A * (B + C) * D \Rightarrow ABC+*D*$

next token	stack	output
none	#	none
A	#	*A*
*	#*	*A*
(	#*(	*A*
B	#*(	*AB*
+	#*(+	*AB*
C	#*(+	*ABC*
)	#*	*ABC+* // unstack!
*	#*	*ABC+**
D	#*	*ABC+*D*
#	empty	*ABC+*D*#* // empty!



isp	icp	operator
	0	(
1	1	unary minus, !
2	2	*, /, %
3	3	+, –
4	4	<, <=, >, >=
5	5	==, !=
6	6	&&
7	7	||
8		(, #

isp: in-stack priority
icp: in-coming priority
#: end of expression

'(' has high priority if it is not in stack
'(' has low priority if it is in stack
-- only ')' can cause it get unstacked

Infix to Postfix – Method 2 (4/4)

```
void Postfix(Expression e) {
// Output the postfix form of the infix expression e.
// NextToken() and stack are as in Eval().
// Assume that the last token in e is '#.' Also, '#' is used at the bottom of the stack
    Stack<Token> stack; // initialize stack
    stack.Push('#');
    for (Token x = NextToken(e); x != '#'; x=NextToken(e))
        if (x is an operand) cout << x;
        else if ( x == ')') { // unstack until '('
            for (; stack.Top() != '('; stack.Pop())
                cout << stack.Top();
            stack.Pop(); // unstack '('
        } else { // x is an operator
            for (; isp(stack.Top()) <= icp(x); stack.Pop())
                cout << stack.Top();
            stack.Push(x)
        }
 // end of expression; empty stack
    for (; !stack.IsEmpty(); cout << stack.Top(), stack.Pop());
    cout << endl;
}
```

$\Theta(n)$,
n is # of tokens in e

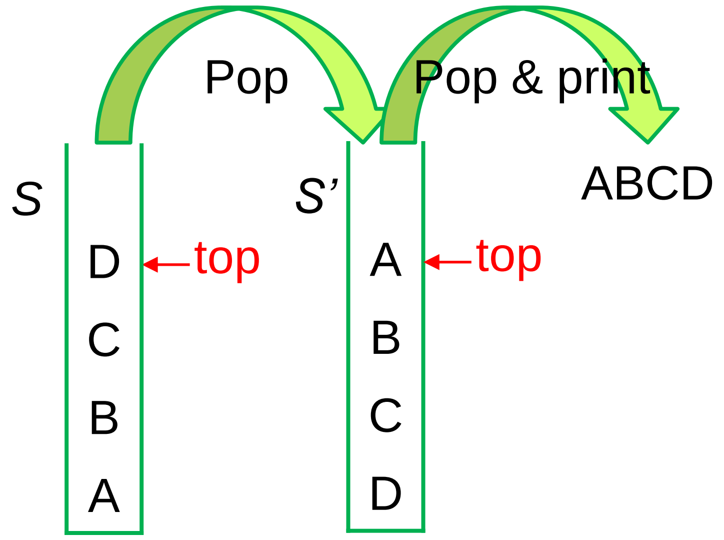
Appendix

How to Print Out a Stack in Reverse Order?

Method 1: Use another stack

1. **Pop *S* into another stack *S'***
2. **Pop *S'* and print out**

- **Example**
  - Input: Stack *S*
  - Output: A B C D



Method 2: Divide-and-conquer

- **Observation:**
  - Print 'E' after "ABCD
  - Recursion!

```
ReverseOut(Stack S)
1.  if S is not empty then do
2.      top ← Pop(S)
3.      ReverseOut(S)
4.      print top
```