CS 2351 Data Structures

Stacks and Queues

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

Outline

- A bag of things and templates in C++
- Stacks
- Queues
- Subtype and inheritance
- Evaluation of expressions

Many Things Are Packed in Bags

A bag of vegetables

A bag of gifts

A bag of tools

A bag of books

Characteristics of "a Bag of X"

- The bag contains objects of type X
- Objects can be added to or deleted from the bag
- The bag may contain multiple occurrences of the same objects, e.g. oranges
- We do not care about the position of an object
- We do not care which object is removed when a delete operation is performed, just taking out one

How to specify "a bag of X" in C++?

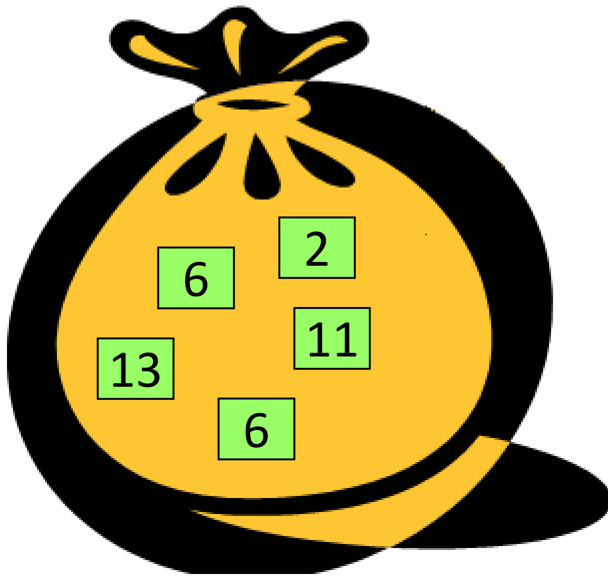
Easy for a Bag of Integers

The func. does not change bag object!

```
class Bag {
 public:
  Bag(int bagCapacity = 10); // Constructor
  ~Bag();                    // Destructor

  int Size() const;          // Return number of elements
  bool IsEmpty() const;      // Check if bag is empty
  int Element() const;       // Return an element in the bag
  void Push(const int);      // Insert an integer into bag
  void Pop()                 // Delete an integer from bag

 private:
  int *array;                // Integer array to store data
  int capacity;              // Capacity of array
  int top;                   // Position of top element
};
```

A Bag of Integers and Its Implementation

A bag of integers



Implementation of
a bag of integers

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
array	6	13	11	6	2			

top: 4

capacity: 8

Element()
Push(20)
Pop()

Implementation of a Bag of Integers

```
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() const { return Size() == 0; }
inline int Bag::Element() const {
   if(IsEmpty()) throw "Bag is empty";
   return array[0];  // Always return the first object
}
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 is empty, cannot delete";
   int deletePos = top/2; // Always delete middle object
   copy(array+deletePos+1, array+top+1, array+deletePos);
   top--;
}
```

Could use different ways

How about a Bag of Floats, Rectangles?

- It is awkward to repeat the same segment of code just to replace **`int`** with **`float`**, **`Rectangle`**, …
- In C++, you can use templates

```
template <class T>
void SelectionSort(T *a, const int n) {
   for (int i = 0; i < n; i++) {
      int j = i;
      // find smallest object 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]);
   }
}
```

Instantiation of Templates

- The template function can be instantiated to the type of the array argument that is supplied to it
  - Operators in templates must be defined for data type T → we cannot use this template for **Rectangle** class unless we overload operator < for **Rectangle**

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

Template for a Bag of X

```
template <class T>
class Bag {
 public:
    Bag(int bagCapacity = 10);  // Constructor
   ~Bag();                     // Destructor
    int Size() const;          // Return number of elements
    bool IsEmpty() const;      // Check if bag is empty
    T& Element() const;        // Return an element in bag
    void Push(const T&);       // Insert an element into bag
    void Pop()                 // Delete an element from bag

 private:
    T *array;                  // Data array
    int capacity;              // Capacity of array
    int top;                   // Position of top element
};
```

Made a constant reference to avoid copy overhead when T is large

Template Implementation for a Bag of X

```
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>
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 is empty, cannot delete";
  int deletePos = top/2; // delete middle emelent
  copy(array+deletePos+1, array+top+1, array+deletePos);
  array[top--].~T();
}
```

Template Implementation for a Bag of X

```
template <class T>
void ChangeSize1D(T*& a, const int oldSize,
                                 const int newSize)
{
   if (newSize<0) throw "New length must be >=0";
   T* temp = new T[newSize];
   int number = min(oldSize, newSize);
   copy (a, a + number, temp);
   delete [] a;
   a = temp;
}
```

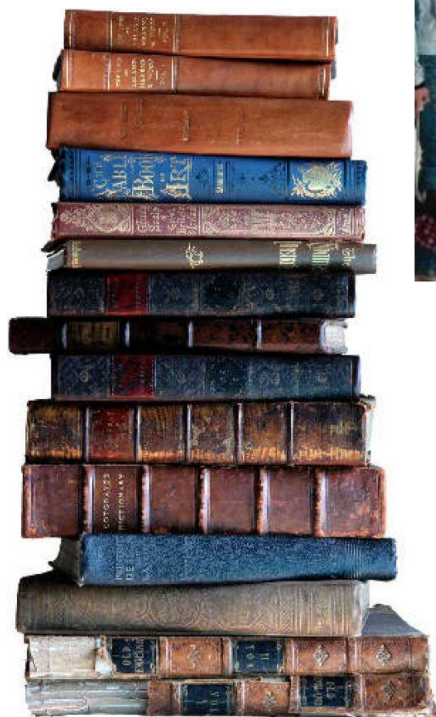
Outline

- A bag of things and templates in C++
- Stacks
- Queues
- Subtype and inheritance
- Evaluation of expressions

A Line of Things

- If we empty "a bag of things" and line them up, then we have "a line of things", c.f. ordered list (fixed indices versus moving positions)

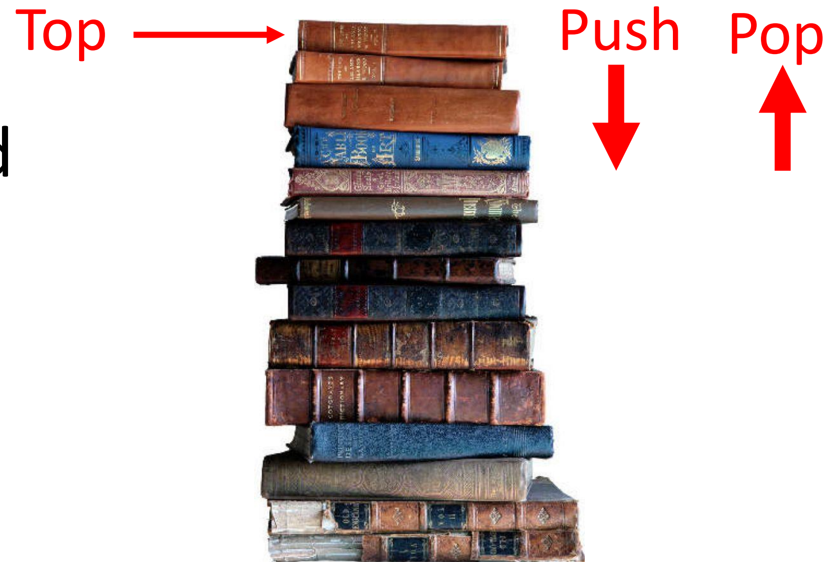
We Encounter a Line of Things Everyday



How to Define “a Line of X” in C++?

There are two types of lines:

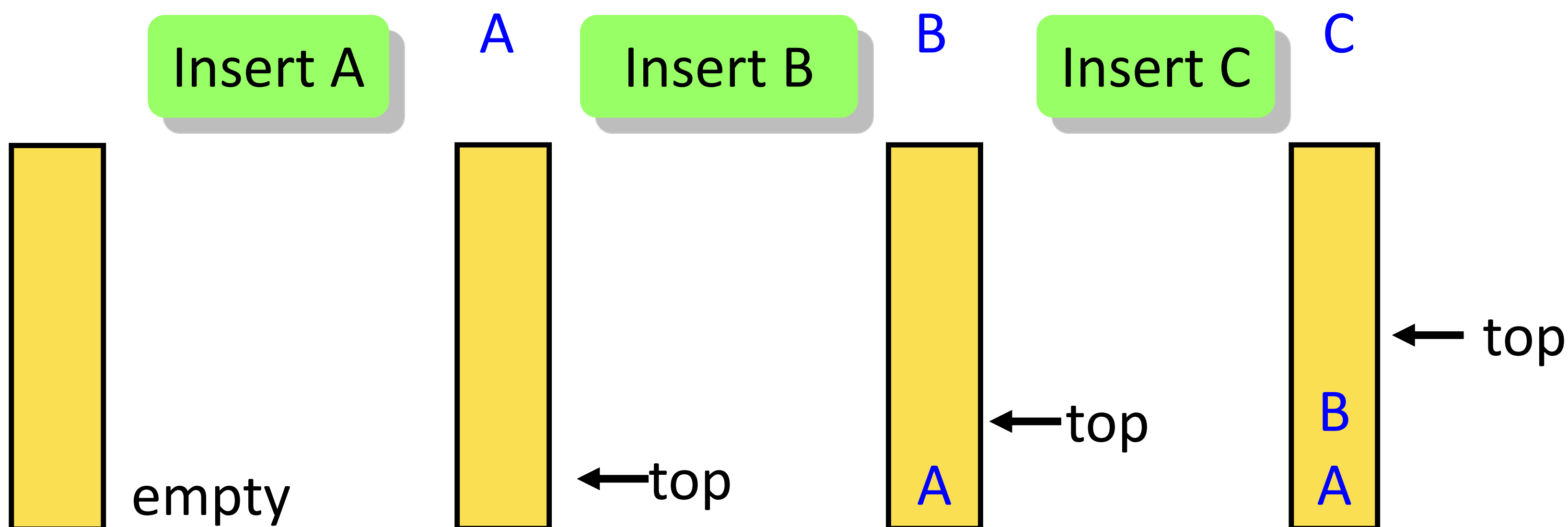
- ***Stack:*** a line that enters and exits at the same end
  - Last-in-first-out (LIFO)



- ***Queue:*** a line that enters at one end and exits at the other
  - First-in-first-out (FIFO)

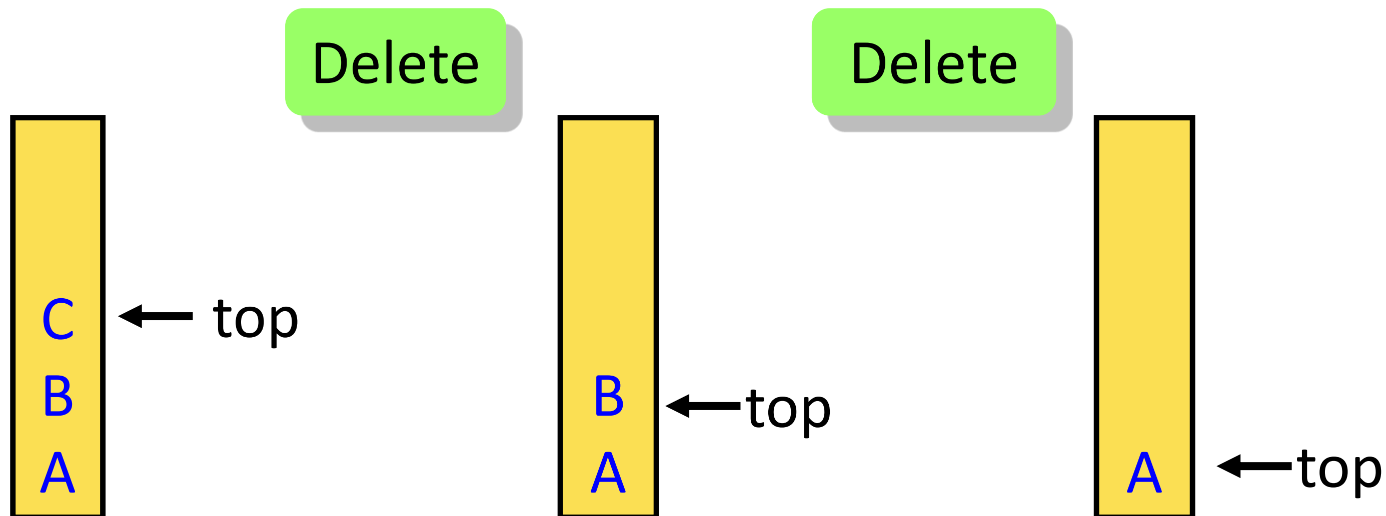
Stack Operations

- Insert a new element into stack (*push*)



Stack Operations

- Delete an element from stack (*pop*)



ADT for a Stack of Things

```
template <class T>
class Stack { // A finite ordered list
public:
      // Constructor
      Stack(int stackCapacity = 10);
      // Check if the stack is empty
      bool IsEmpty( ) const;
      // Return the top element
      T& Top( ) const;
      // Insert a new element at top
      void Push(const T& item);
      // Delete one element from top
      void Pop( );
private:
       T* stack;
       int top;
       int capacity;
};
```

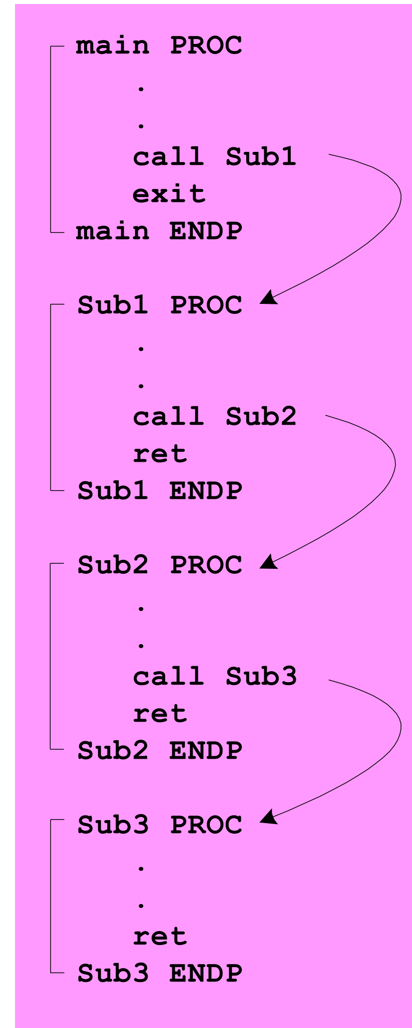
Stack Operations: Push & Pop

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

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

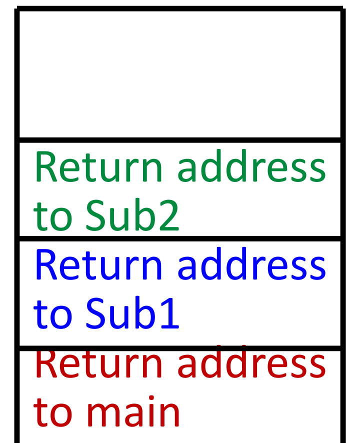
Stack Application

- System stack for function recursion
  - Used at run time to process **recursive function calls** (function calls are LIFO)
  - For each invocation, store function parameters, local variables, and **return address** of the caller function



By the time Sub3 is called, stack contains all 3 return addresses:

System stack

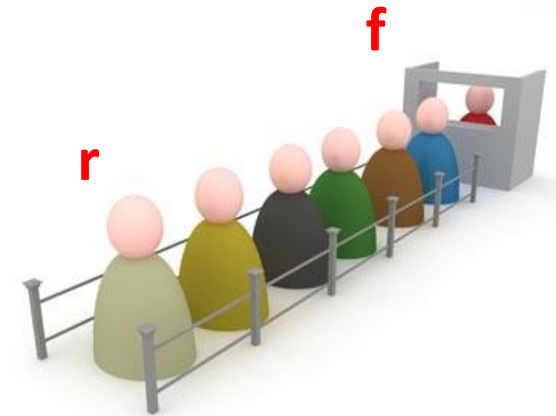


Outline

- A bag of things and templates in C++
- Stacks
- Queues
- Subtype and inheritance
- Evaluation of expressions

Queue Operations

- Insert a new element into queue
  - f: front (where old objects are deleted)
  - r: rear (where next new object is to be inserted)



Insert A

Insert B

Insert C

f r

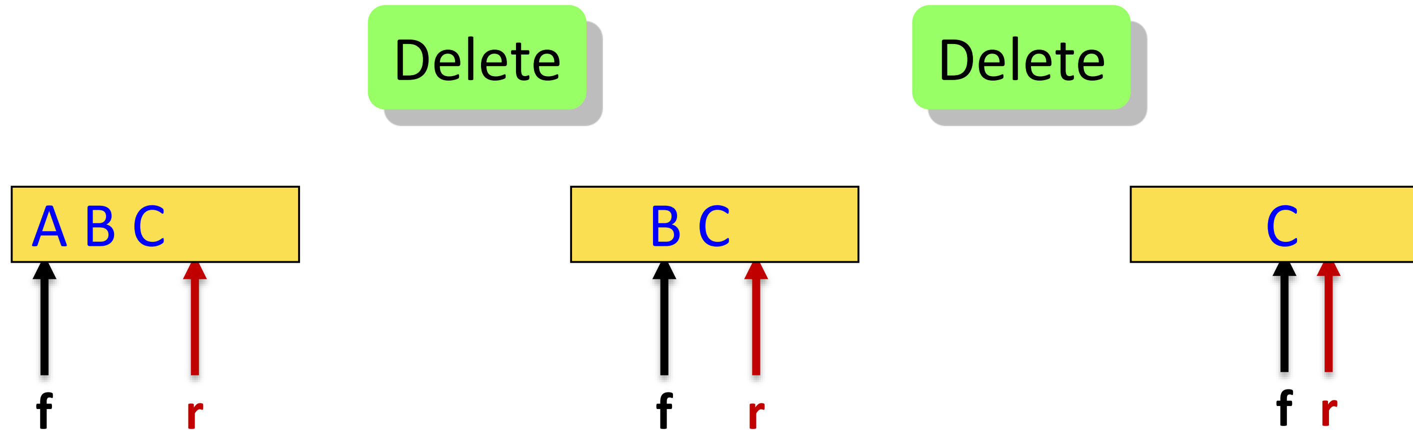
f r

f r

f r

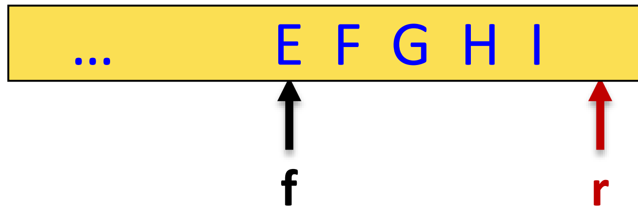
Queue Operations

- Delete an old element from queue
  - f: front (where old objects are deleted)
  - r: rear (where next new object is to be inserted)

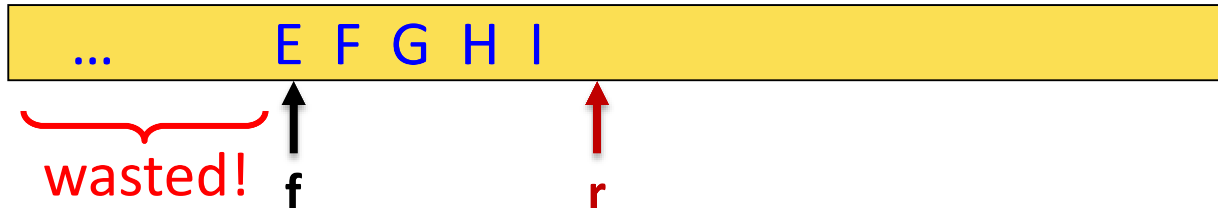


Problem

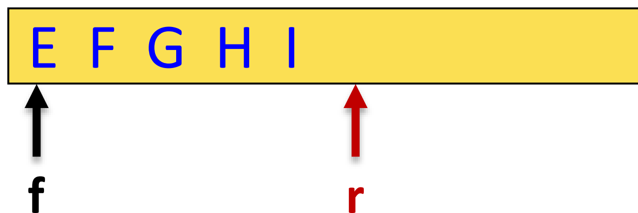
- What happen if (rear == capacity-1) ?



- Add more space?



- Shift left?



Complex code, extra time

Circular Queue

rear rear rear

C B A | C D B A | C D B

front front front

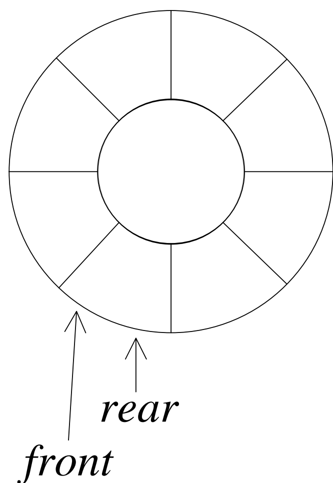
front: 1 position ahead of 1st object
rear: last object

Initial Insertion Deletion

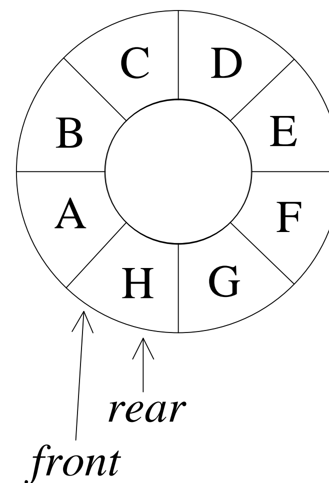
```
rear = (rear+1) % capacity;
```

Circular Queue

- When is the queue empty?
  - rear == front?



Queue is empty



Queue is full

Allocate additional space before the queue is full

ADT of Queue

```
template <class T>
class Queue { // A finite ordered list
  public:
      Queue(int queueCapacity = 10);
      // Check if the stack is empty
      bool IsEmpty( ) const;
      // Return the front element
      T& Front( ) const;
      // Return the rear element
      T& Rear( ) const;
       // Insert a new element at rear
      void Push(const T& item);
      // Delete one element from front
      void Pop( );
  private:
       T* queue;
       int front, rear;
       int capacity;
};
```

Queue Operations

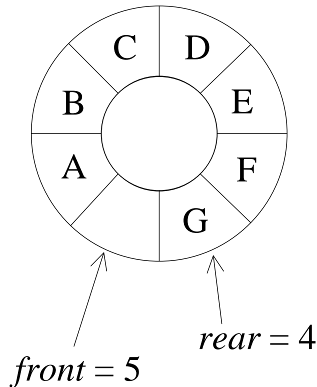
```
template <class T>
void Queue <T>::IsEmpty() {return front==rear;}

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

template <class T>
T& Queue <T>::Rear() {
    if (IsEmpty()) throw "Queue is empty!";
    return queue[rear];
}
```

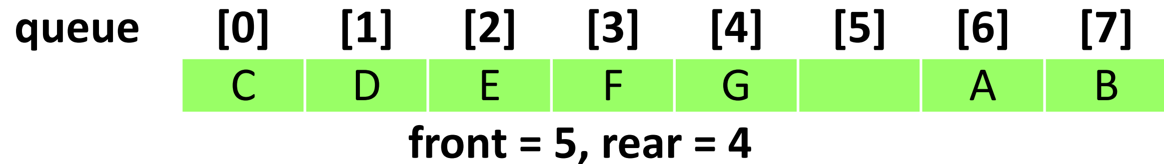
Queue Operations: Push & Pop

```
template <class T>
void Queue <T>::Push(const T& x)
{   // Add x at rear of queue
    if((rear+1)%capacity == front) {
    // queue is going to full, double the capacity!
    }
    rear = (rear+1)%capacity;
    queue[rear] = x;
}
```

```
template <class T>
void Queue <T>::Pop( )
{   // Delete front element from queue
    if(IsEmpty()) throw "Queue empty, cannot delete";
    front = (front+1)%capacity;
    queue[front].~T(); // Delete the element
}
```

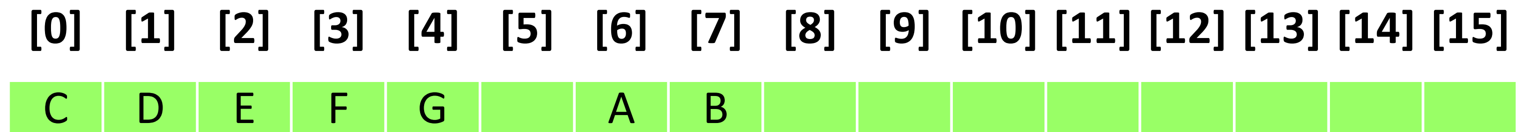
Doubling Queue Capacity





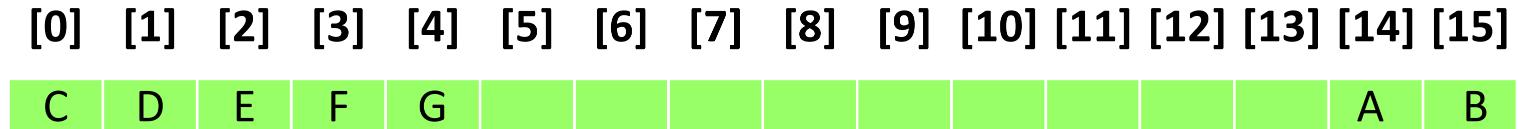
front = 5, rear = 4

Expanded full circular queue



front = 5, rear = 4

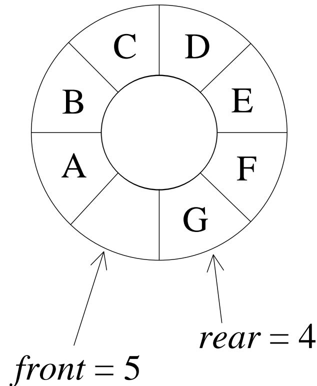
Doubling the array



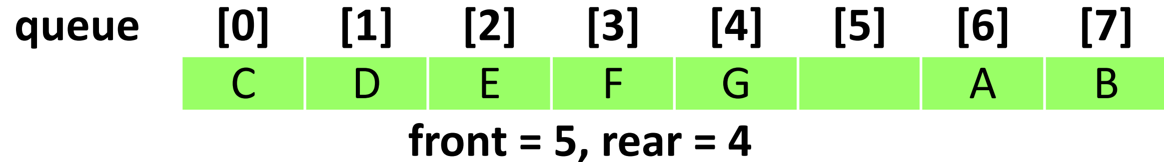
front = 13, rear = 4

Scenario 1: After shifting right segment

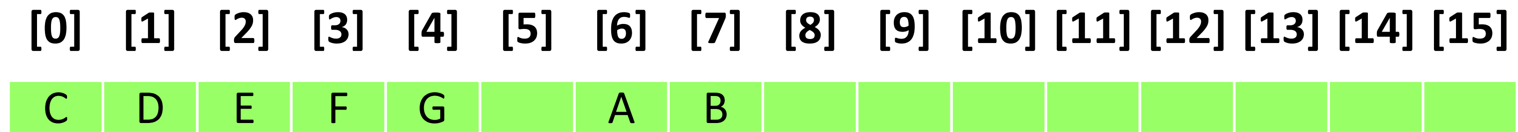
Doubling Queue Capacity



Full circular queue

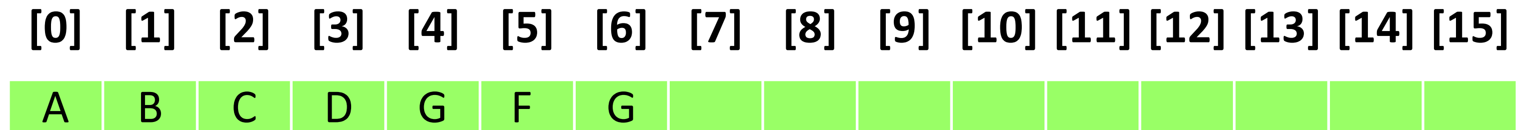


Expanded full circular queue



front = 5, rear = 4

Doubling the array



front = 15, rear = 6

Scenario 2: Alternative configuration

Outline

- A bag of things and templates in C++
- Stacks
- Queues
- Subtype and inheritance
- Evaluation of expressions

Bag vs Stack

- Is a stack a bag?

- Is a bag a stack?

C++ Bag vs C++ Stack

```
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();
};
```

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


  bool IsEmpty() const;
  T& Top() const;

  void Push(const T& item);
  void Pop();
};
```

Subtype and Inheritance

- We say that **`Stack`** is derived/inherited from **`Bag`**
  - **`Bag`** is the *base class*
  - **`Stack`** is the *derived class*
  - **`Stack`** is a *subtype* of **`Bag`**
- All member functions in **`Stack`** are same as those in **`Bag`** except **`Top()`** and **`Pop()`**

Subtype Definition in C++

```
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;
};
```

Implement operations different from or not in Bag class

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

Notes on Inheritance

- The derived class **`Stack`** inherits all the members (data and functions) of the base class **`Bag`**
  - Only the non-private members of the base class are accessible to the derived class
  - Inherited public and protected members of the base class have the same level of access in derived class
- The member functions inherited by the derived class have the same prototype → *interface reuse*
  - The implementation can be reused, but can also be overridden, e.g. Pop()
  - Constructor and destructor cannot be inherited

Bag vs Queue

- **Queue** can also be represented as a derived class of **Bag**, but the implementations of **Bag** and **Queue** are less similar

```
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()
};
```

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


   bool IsEmpty() const;
   T& Rear() const;
   T& Front() const;

   void Push(const T& item);
   void Pop();
};
```

Outline

- A bag of things and templates in C++
- Stacks
- Queues
- Subtype and inheritance
- Evaluation of expressions
  - An example of using stack

An Arithmetic Expression

$$X = A / B - C + D * E - A * C$$

- Operators: +, -, *, /, …
- Operands: A, B, C, D, E, F

Expression Evaluation

- For X = A/B – C + D*E – A*C
- If A=4, B=C=2, D=E=3
- X = ((4/2)-2)+(3*3)+(4*2)=1

- For X = (A/(B – C + D))*(E – A)*C
- If A=4, B=C=2, D=E=3
- X = (4/(2-2+3))*(3-4)*2 = -2.6666666

The order of applying the operations are important!

Evaluation Rules

- Operators have **priority**
- Operator with **higher priority** is evaluated first
- Operators of **equal priority** are evaluated from **left to right** → *left associative*
  - **`a/b*c`**: For b, which operator is evaluated first, / or *?
- **Unary** operators are evaluated from **right to left**
  - **`a=b=c=2`**

Priority of Operators in C++

Priority	Operators
1	Minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	= =, !=
6	&&
7	||

Infix and Postfix Notation

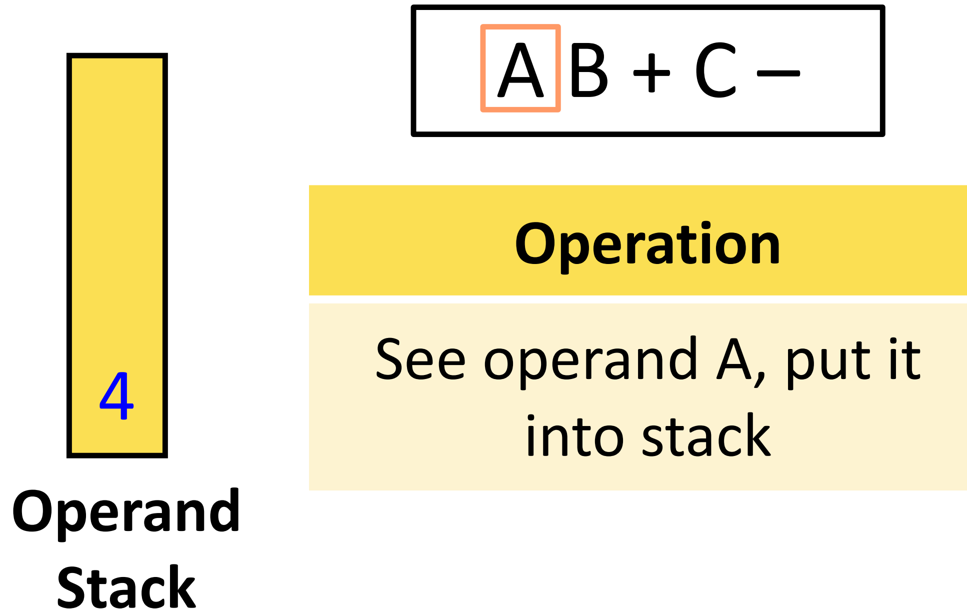
- **Infix** notation
  - Operator comes in between the operands, e.g. A+B*C
  - Hard to generate machine code
- **Postfix** notation
  - Each operator appears after its operands, e.g. ABC*+
  - No need for **parentheses**
  - Operator priority is irrelevant and given in the expression
  - Efficient evaluation using stack

Infix to Postfix Evaluation

- Phase 1: Infix to postfix conversion
  6/2-3+4*2 → 6 2 / 3 – 4 2 * +
- Phase 2: Postfix expression evaluation
  6 2 / 3 – 4 2 * + → 8
  - Making a left to right scan
  - Putting operands into stack
  - On encountering an operator, popping operands and evaluating
  - Pushing the result back into stack

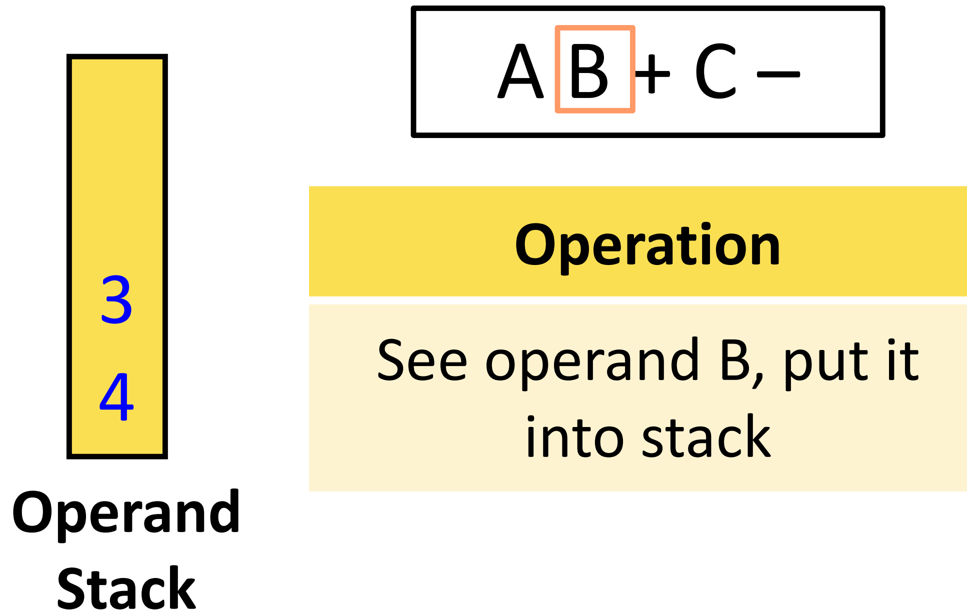
Example of Postfix Expression Evaluation

- Infix: A+B – C ➔ Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2



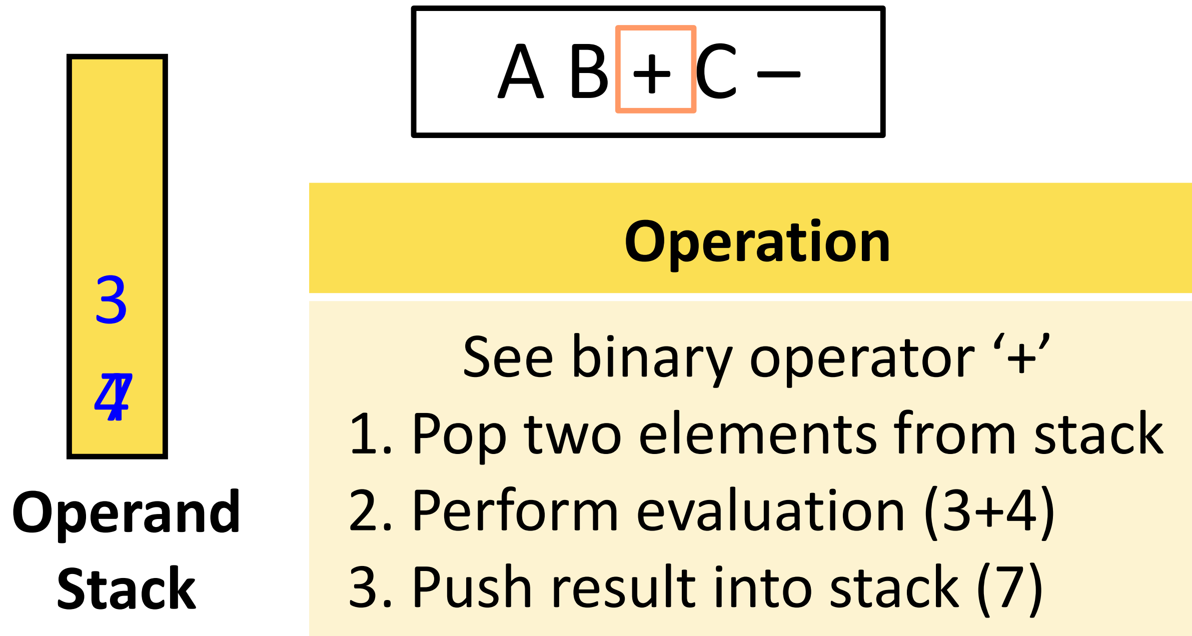
Example of Postfix Expression Evaluation

- Infix: A+B – C ➔ Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2



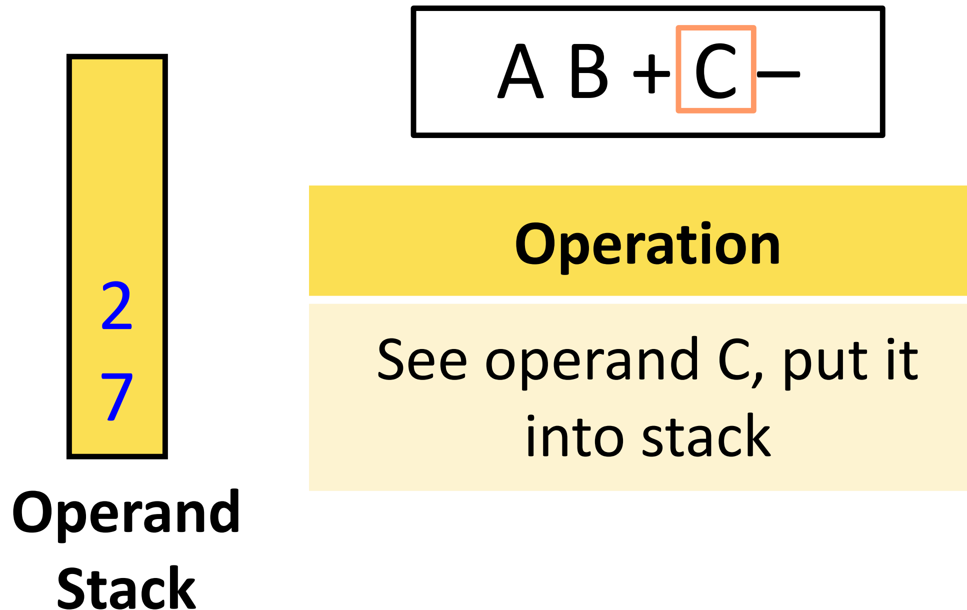
Example of Postfix Expression Evaluation

- Infix: A+B – C ➔ Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2



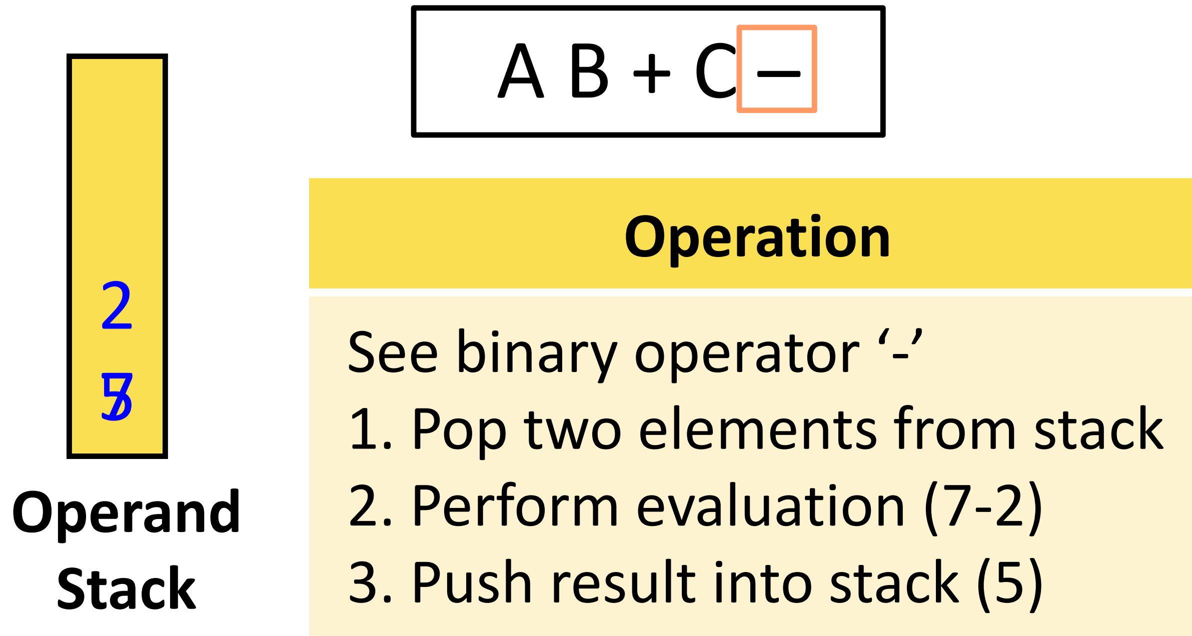
Example of Postfix Expression Evaluation

- Infix: A+B – C ➔ Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2



Example of Postfix Expression Evaluation

- Infix: A+B – C ➔ Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2



Evaluation of Postfix Expressions

```
void Eval(Expression e)
{  // Assume the last token of e is '#'
   // NextToken() gets next token in
   Stack<Token> stack; // initialize stack
   for (Token x=NextToken(e); x != '#';
                              x=NextToken(e)){
      if(x is an operand) stack.Push(x);
      else{
      // Pop correct # of operands from stack
      // Perform the operation x
      // Push the result back to stack
      }
   }
};
```

Machine Code Generation for Expressions

- Phase 1: Infix to postfix conversion

  `a/b - c + d*b → a b / c - d b * +`

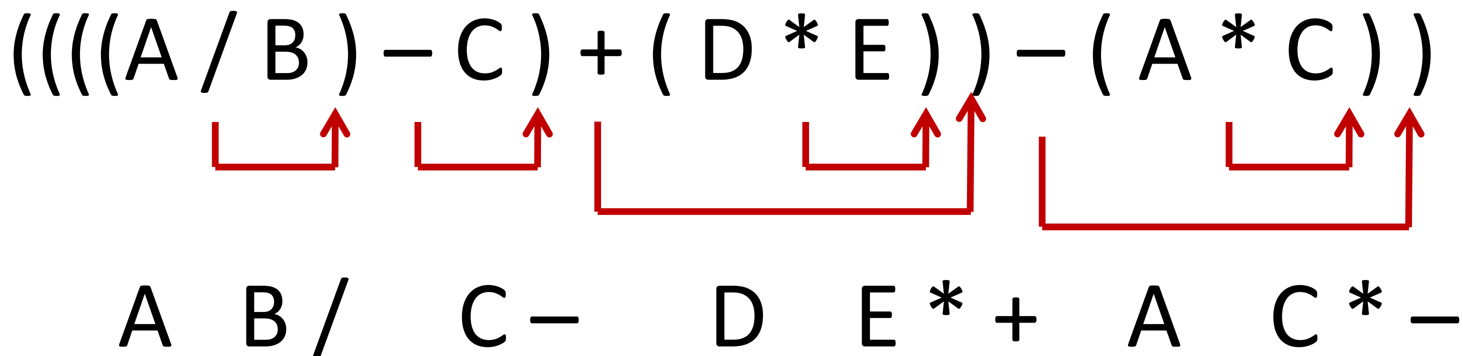
- Phase 2: Postfix expression evaluation

`a b / c - d b * +`

```
load r1,M[a]
load r2,M[b]
div  r3,r1,r2
load r4,M[c]
sub  r5,r3,r4
load r6,M[d]
mult r7,r2,r6
add  r8,r5,r7
```

Infix to Postfix Conversion

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



Infix to Postfix

- Smarter algorithm
  - Scan the expression only once
  - Utilize stack
- The order of operands dose not change between infix and postfix
  - Output every visited operand directly
- Use stack to store visited operators and pop them out at the right moment
  - When the ***priority*** of the operator on top of stack is ***higher or equal to*** that of the incoming operator (left-to-right associativity)

Example 1

- Infix: A + B * C

Next token	Stack	Output
None	Empty	None
A	Empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
	+	ABC*
	Empty	ABC*+

Example 2

- Infix: A * (B + C) * D

Next token	Stack	Output
None	Empty	None
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
	Empty	ABC+*D*

Notes

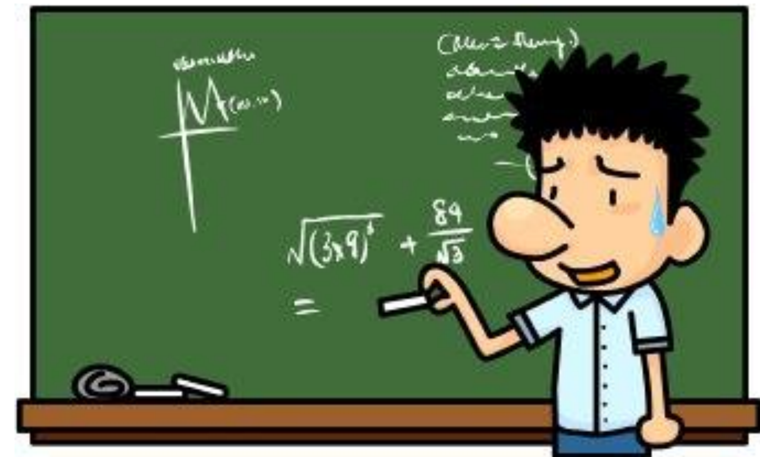
- Expression with ( )
  - ‘(‘ has the highest priority, always push to stack
  - Once pushed, ‘(’ get lowest priority
  - Pop the operators until you see the matching ‘)’

Try It Out …

- Find the postfix representation of the following infix expression:

  `a * (b + c) – a * d * e`

- Suppose a=1, b=2, c=3, d=4, e=5 and we use postfix to evaluate the expression. Draw the content of the stack when the second * is encountered



Summary

- Template, subtype, inheritance in C++
- Stacks are last-in-first-out
- Queues are first-in-first-out
  - Circular queues
- Evaluation of expressions as an example of using stacks
  - Infix to postfix conversion
  - Evaluation of postfix expressions