

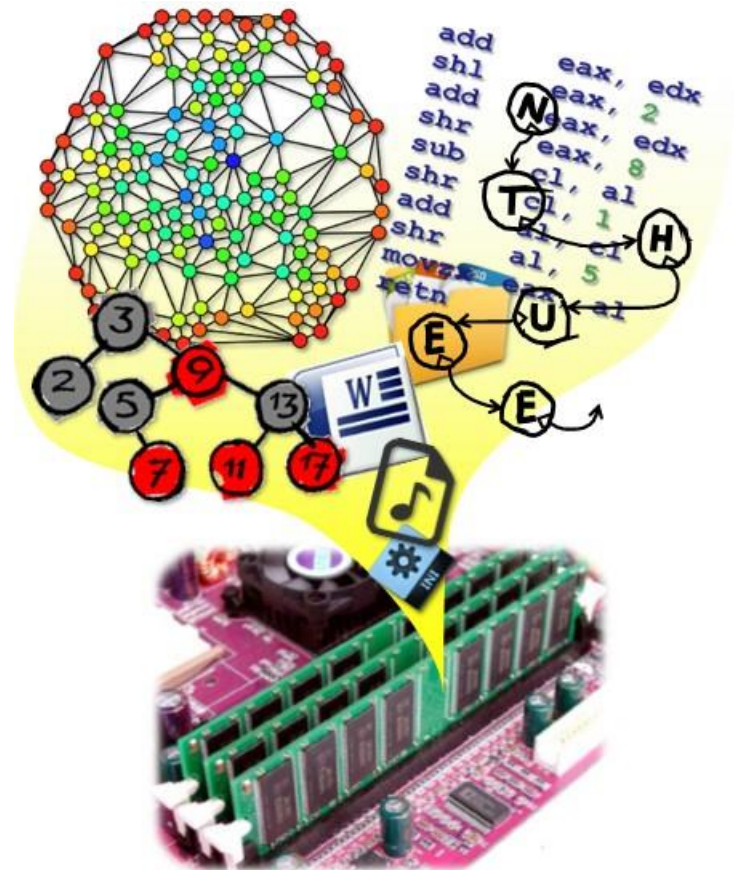
Data Structures

CH3 Stacks & Queues

Prof. Ren-Shuo Liu

NTHU EE

Spring 2017





Outline

- **3.1 Templates in C++**
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- 3.6 Evaluation of expressions



Observations

- Many codes look the same for different **types**
 - Sorting **functions** that handle
 - 32-bit integers
 - 64-bit integers
 - float
 - ...
 - Sparse matrix **classes** that handle
 - 32-bit integers
 - 64-bit integers
 - float
 - ...



Non-Template Solutions

- Implement the same behavior over and over
 - Hard to maintain code
 - Hard to globally modify code
- Write general code for a common base type
 - Lose the benefits of compiler's type checking
 - Incurs overhead
- Use macros (`#define`)
 - Sacrifice readability
 - Sacrifice debuggability



Template

- Template can be instantiated to any data type
 - So called "parameterized types"
- C++ language supports
 - Template functions
 - Template classes



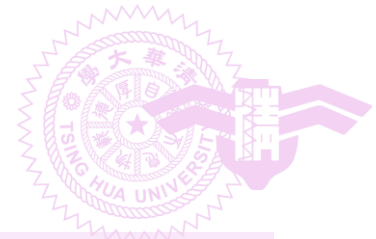
Template Function Example

```
void SelectionSort (int *a , const int n )  
{  
    for (int i = 0 ; i < n ; i++ )  
    {  
        int j = i;  
        for ( int k = i + 1 ; k < n ; k++ )  
            if ( a[k] < a[j] ) j = k;  
        swap ( a[i], a[j] );  
    }  
}
```



```
template <class T>  
void SelectionSort (T *a , const int n )  
{  
    for (int i = 0 ; i < n ; i++ )  
    {  
        int j = i;  
        for ( int k = i + 1 ; k < n ; k++ )  
            if ( a[k] < a[j] ) j = k;  
        swap ( a[i], a[j] );  
    }  
}
```

- template <class T> is identical to template <typename T>
- It is a convention to use "T", but one can use any other name



Bag Class (for integers)

```
class Bag
{
public:
    Bag ( int bagCapacity = 10 );    // constructor
    ~Bag( );                        // destructor
    int Size( ) const;              // return number of elements in bag
    int Element( ) const;           // return an element that is in the bag
    void push(const int);           // add an integer into the bag
    void pop();                     // delete an integer in the bag

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

const member function

Specifies that the function does not modify the object for which it is called.

```
const Bag emptyBag;
emptyBag.size(); //valid
emptyBag.push(1); //error
```



Bag Class (for integers)

```
Bag::Bag (int bagCapacity)
:capacity ( bagCapacity )
{
    if ( capacity < 1 )
        throw "Capacity must be > 0";

    array = new int [ capacity ];
    top = -1;
}

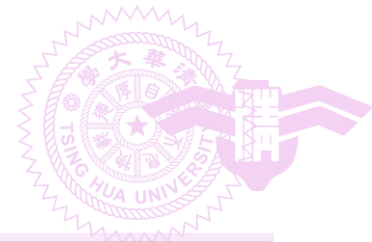
Bag::~~Bag ( )
{ delete [] array; }

inline int Bag::Size( ) const
{ return top + 1; }
```

Initialization list



initialize member variables
when they are created
rather than afterwards

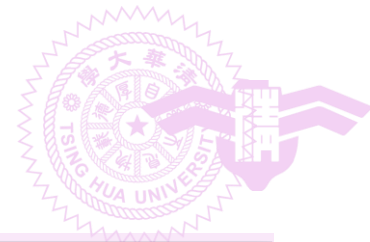


Bag Class (for integers)

```
inline int Bag::Element ( ) const
{
    if ( IsEmpty ( ) )
        throw "Bag is empty";

    return array [0]; // always return 0th element
}

void Bag::Push (const int x)
{
    if (capacity == top + 1) {
        ChangeSize1D (array, capacity, 2 * capacity);
        capacity *= 2;
    }
    array[++top] = x;
}
```

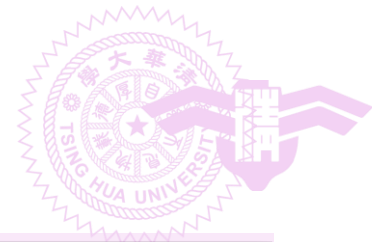



Template Bag

```
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>
inline int Bag<T>::Size( ) const
{ return top + 1; }
```



Template Bag

```
template <class T>
inline int Bag<T>::Element( ) const
{
    if ( IsEmpty() )
        throw "Bag is empty";
    return array [0];
}

template <class T>
void Bag<T>::Push(const T x)
{
    if (capacity == top + 1) {
        ChangeSize1D (array, capacity, 2 * capacity);
        capacity *= 2;
    }
    array [++top];
}
```



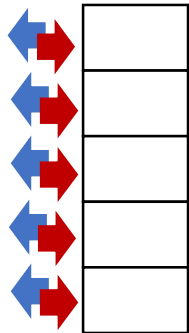
Outline

- 3.1 Templates in C++
- **3.2 The stack ADT**
- **3.3 The queue ADT**
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- 3.6 Evaluation of expressions



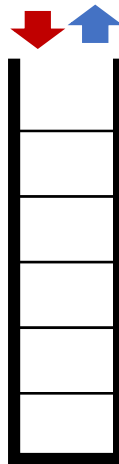
Stacks and Queues

- Two frequently used data structures
- They are special cases of the more general data structure type, lists



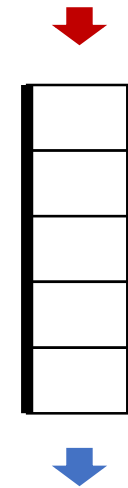
List

(Random add/del)



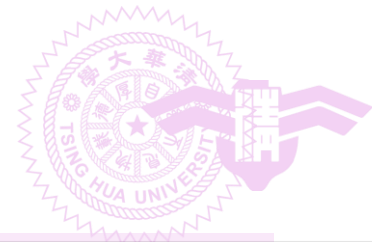
Stack

(Last In First Out)



Queue

(First In First Out)



Stack and Queue ADTs

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

    bool IsEmpty( ) const;

    void Push(const T& item);
    // add an item into the stack

    void Pop( );
    // delete an item
    ...

};
```

```
template < class T >
class Queue
{
public:
    Queue (int queueCapacity = 0);

    bool IsEmpty( ) const;

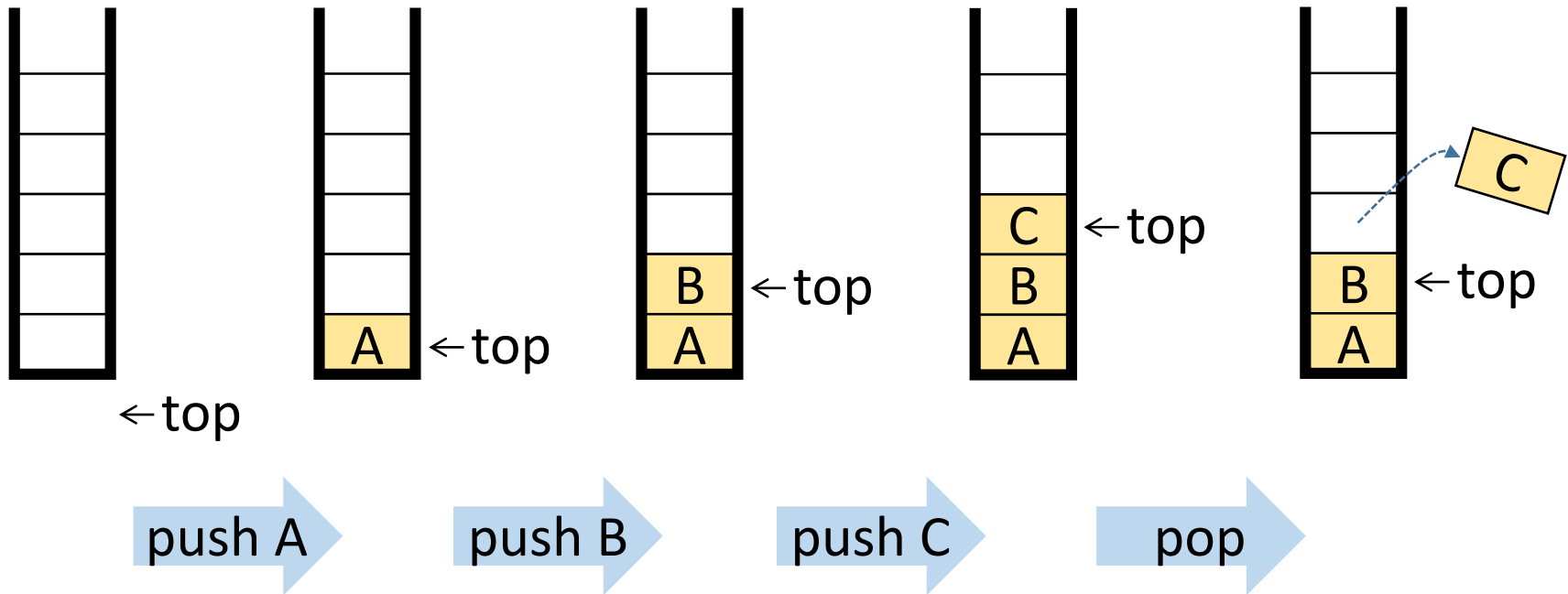
    void Push(const T& item);
    // add an item into the queue

    void Pop( );
    // delete an item
    ...

};
```



Stack

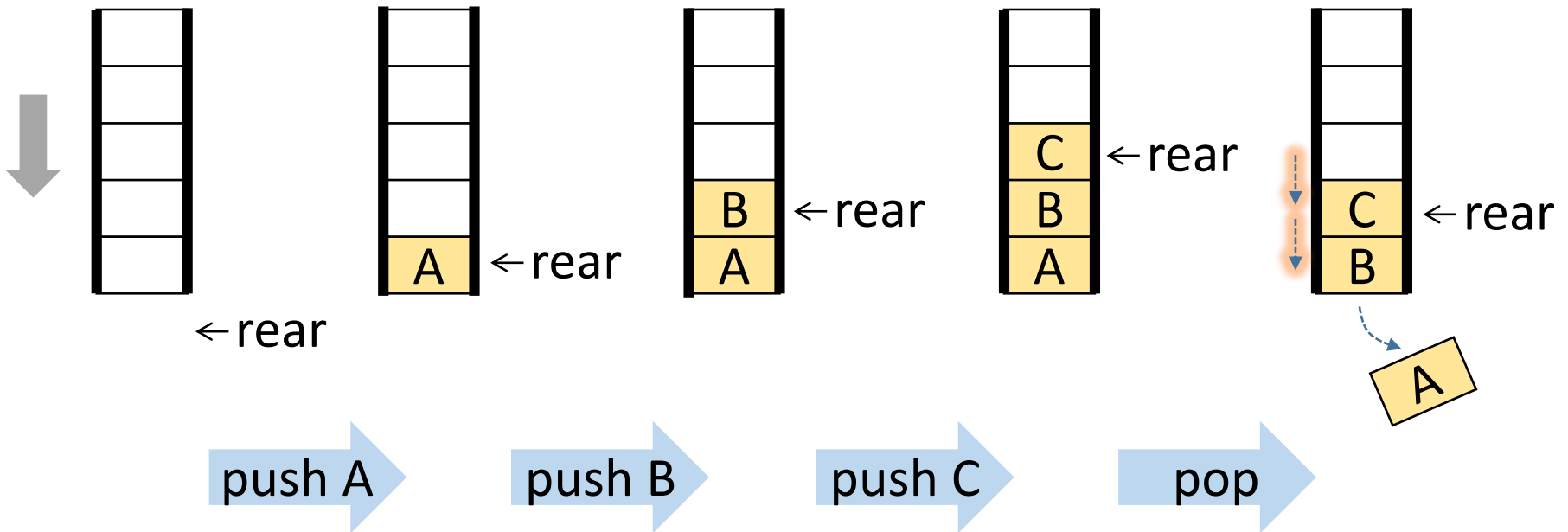


Time complexity

- $\text{push}(): \Theta(1)$
- $\text{pop}(): \Theta(1)$



Queue (Single Pointer)

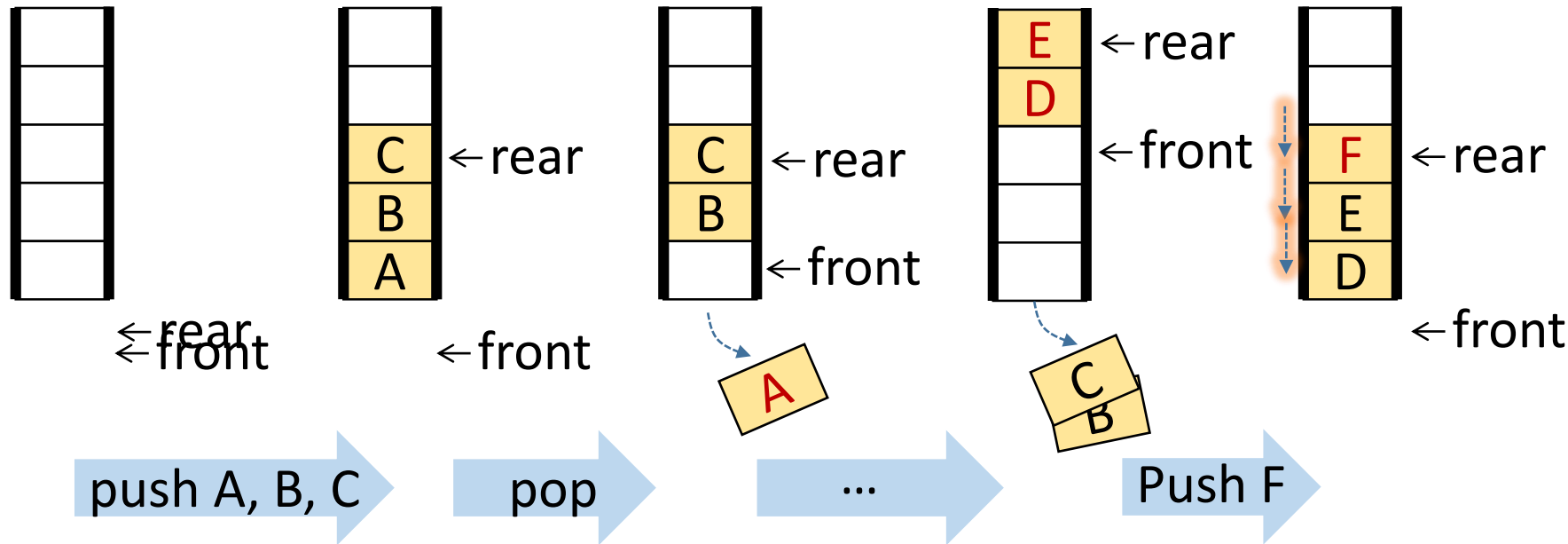


Time complexity

- push(): $\Theta(1)$
- pop(): $\Theta(\text{size})$



Queue (Dual Pointers)



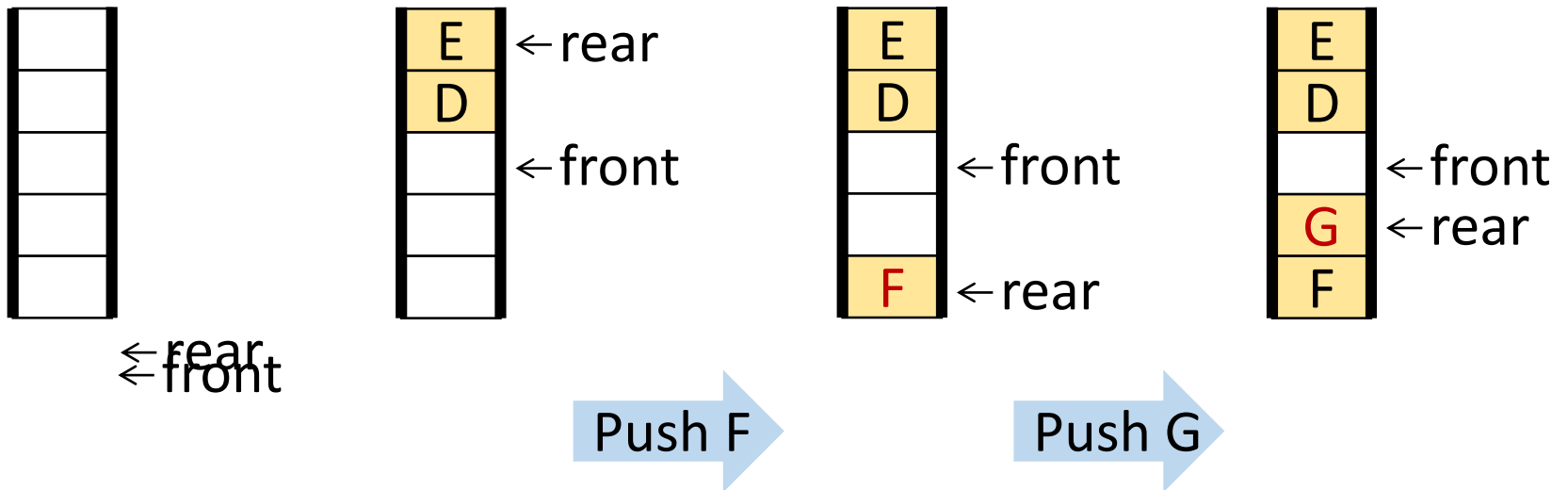
Time complexity

- $\text{push}(): O(\text{size})$
 - When the rear pointer reaches the boundary and a push occurs, data need to be moved
- $\text{pop}(): \Theta(1)$



Circular Queue

- Permit the queue to **wrap around** the end space

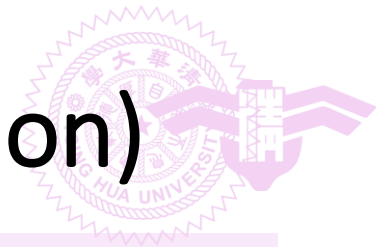


Time complexity

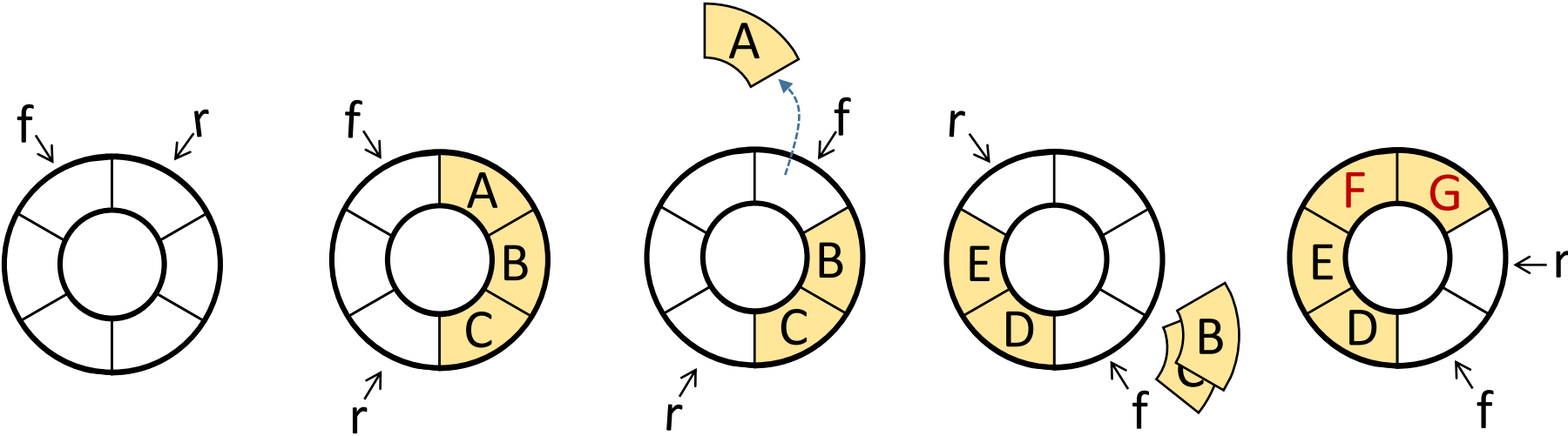
- push(): $\Theta(1)$
- pop(): $\Theta(1)$

Note that in this version of circular buffer, the position that the front pointer points to is a dead space. A slot is deliberately unused.

- Otherwise, we cannot determine whether the queue is empty or full.



Circular Queue (Circular Illustration)



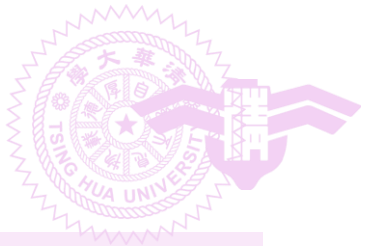
push A, B, C → pop → ... → push F, G



Outline

- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- **3.4 Subtyping and inheritance in C++**
- 3.6 Evaluation of expressions
- 3.5 A mazing problem

Relationships Between Things



- We abstract things on two key dimensions
 - IS-A relationship
 - HAS-A relationship
- Real world examples
 - iPhone **is a** smartphone. iPhone **has a** battery
 - NTHU **is a** university. NTHU **has a** Math department
- ADT examples
 - Rectangle **is a** Polygon. Rectangle has a *height* dimension
 - Stack **is a** Bag. Stack **has a** *top* pointer
 - Stack is a specialized bag that requires elements to be deleted in the LIFO order



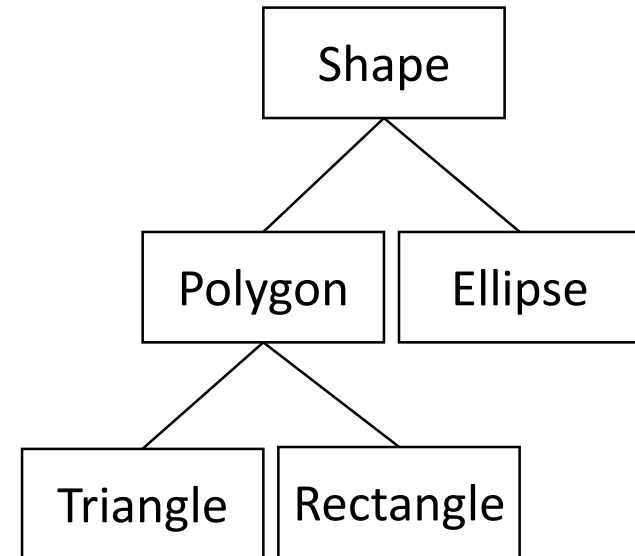
Subtype / IS-A / Subclass

- Subtype
 - Equivalent concept to the **IS-A** relationship
 - Rectangle **is a subtype of** Polygon
 - Since C++ use *classes* to denote data types, subtypes are also widely referred to as **subclasses**
- Subtype is conceptual relationship between ADT **specifications**
 - "Stack IS A Bag" is true regardless of the implementation



Inheritance

- Use
 - Express **IS-A** relationships between classes
 - Derive a new class (**derived class / sub type / sub class**) from an existing class (**base class**)
- Objective
 - Eliminate redundant implementation
 - Members (data and functions) are by default inherited from a base class to a derived class
- Different inheritance styles
 - **Public** inheritance
 - **Access levels** (public/protected/private) of the members are also inherited
 - **Protected** inheritance
 - **Private** inheritance





Effects of Inheritance

- Stack inherits from Bag
 - Stack must redefine its constructors and destructors
 - Stack can redefine its unique data and functions (pop and top)
 - Stack inherits all the other data and functions of Bag

```
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 top;
};
```

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



Usage Example of Derived Classes

```
Bag b(4); // invoke Bag constructor
Stack s(7); // invoke Stack constructor, which also invokes Bag constructor
b.Push(2017); // use Bag::Push()
s.Push(330); // Stack does not contains a specialized Push(), so use Bag::Push
b.Pop(); // use Bag::Pop()
s.Pop(); // Stack contains a specialized Pop() overriding Bag::Pop(), so use Stack::Pop()
```

```
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 top;
};
```

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

Syntax of Implementing Derived Classes



```
Stack::Stack(int stackCapacity)
: Bag(stackCapacity)
// explicitly call to the Bag constructor that has arguments
{
    // here is code specifically for creating a stack, if any
}
int Stack::Stack( )
{
    // here is code specifically for destroying a stack, if any
}
//Bag destructor is automatically called when a stack is destroyed

int Stack::Top( ) const
{
    if (IsEmpty( )) throw "Stack is empty.";
    return array[top];
}
void Stack::Pop( )
{
    if (IsEmpty( )) throw "Stack is empty. Cannot delete.";
    top--;
}
```



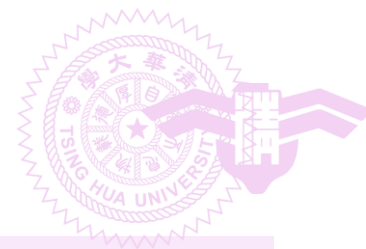
Outline

- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- **3.6 Evaluation of expressions**
- 3.5 A mazing problem



Evaluation of Expressions

- Arithmetic expressions
 - $X = (A / B) - C + D * E - A * C$
- Boolean expressions
 - $X = (A == B) || !(C > D)$
- Expressions are made up of
 - **Operands**: A, B, C, D, E
 - Binary arithmetic **operators**: +, -, *, /, %
 - Unary arithmetic **operators**: -
 - Relational **operators**: <, <=, ==, !=, >=, >
 - Binary logical **operators**: &&, ||
 - Unary logical **operators**: !
 - **Delimiters**: (,)



Evaluation of Expressions

- Let's focus on an arithmetic expression
 - $X = A / B - C + D * E - A * C$
- Order of operations matter
 - Let $A = 4, B = C = 2, D = E = 3$
 - $((4/2)-2)+(3*3)-(4*2) = 0 + 9 - 8 = 1$
 - $(4/(2-2+3))*(3-4)*2 = (4/3)*(-1)*2 = -2.666...$
- How can computers uniquely define the order of an expression?



Priority of Operators

- Priority is introduced to help defining the order

	Priority	Operator
High	1	Unary minus (負號), !
	2	*, /, %
	3	+, -
	4	<, <=, >=, >
	5	=, !=
	6	&&
	7	

} * and / have higher priority than + and -

- Tie break rule: left to right

- Example Two operators compete for one operand

• $A / B - C + D * E - A * C \rightarrow (A/B) - C + (D * E) - (A * C)$

*Note: In the original image, blue circles highlight the operators /, +, and * in the expression, and green arrows point from the text 'Two operators compete for one operand' to the / and + operators, and from the text ''/' and '*' win' to the / and * operators.*

• $A/B * C / D \rightarrow ((A/B) * C) / D$ Tie-break rule



Infix and Postfix Notations

- Infix
 - Binary operators come **in-between** their operands
 - $A*B/C$
- Postfix
 - Binary operators appear **after** their operands

- Examples

- Infix:

$A*B$

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

- Postfix:

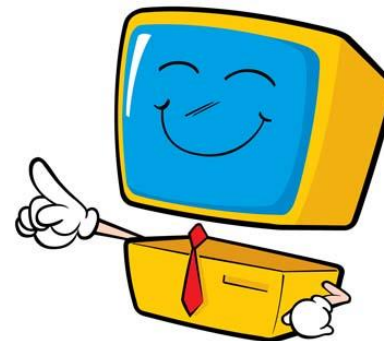
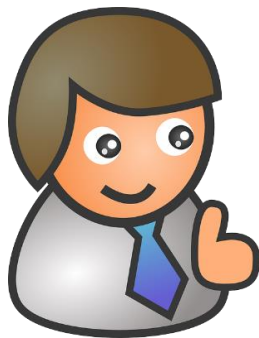
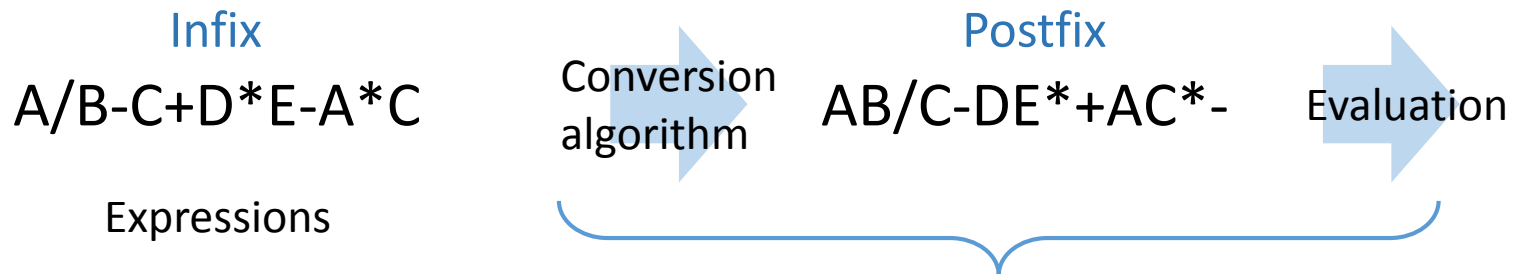
AB^*

$AB/C-DE^*+AC^*-$



Two Essential Algorithms

- Combining two algorithms enables computers to handle human-written expressions
 - Infix-to-Postfix conversion
 - Postfix evaluation (just mentioned)

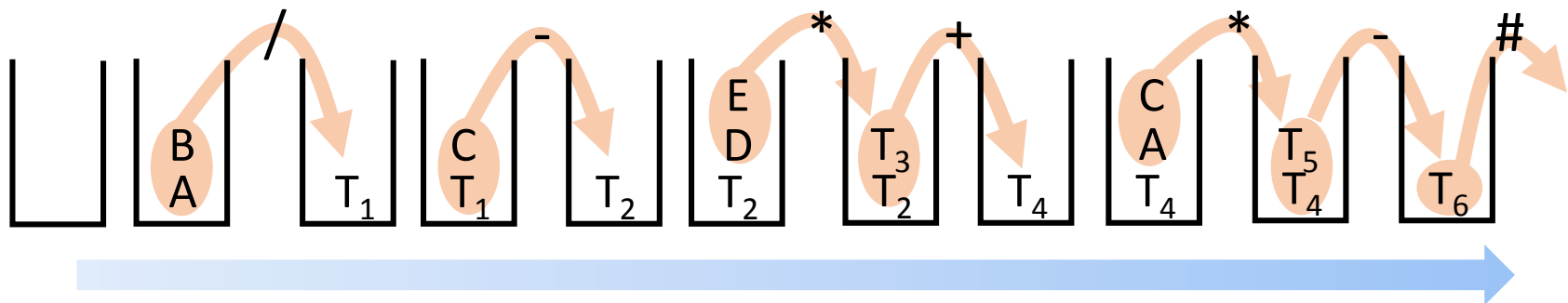


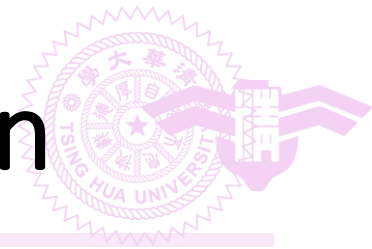


Postfix Evaluation

- Rules
 - Left to right scan
 - Push operands onto a stack
 - Evaluate operators using the required number of operands from the stack
 - Push the evaluating results onto the stack again

• $AB/C-DE^*+AC^*-\#$ (*# denotes the end of an expression*)





Advantages of Postfix Notation

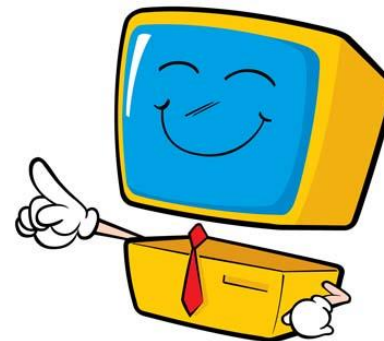
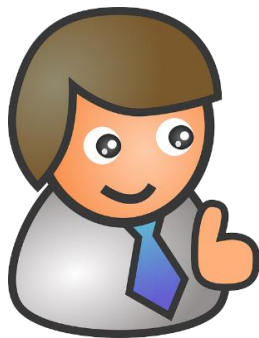
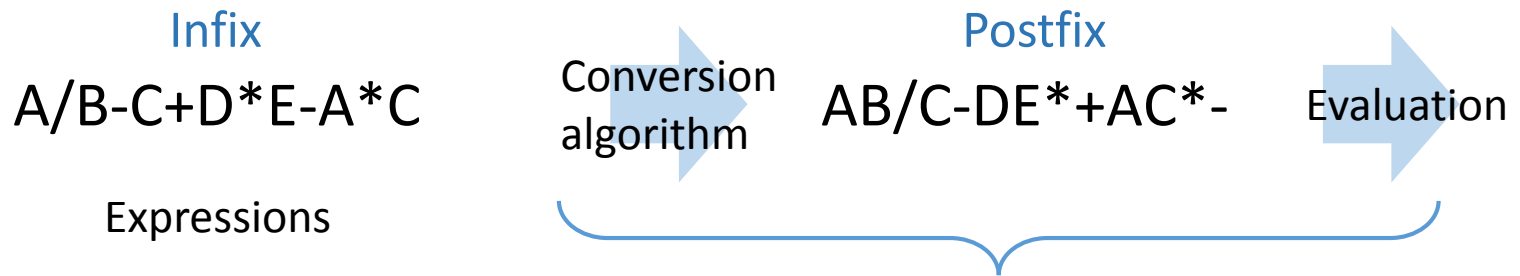
- Evaluation is **simpler** than infix notation
 - The need for **parenthesis** is gone
 - The need for **operator priority** is gone

```
void Eval(Expression e)
{
    Stack<Token> stack; // initialize a stack
    for (Token x = NextToken(e); x!= end of expression; x=NextToken(e))
    {
        if (x is an operand) {
            stak.Push(x)
        } else { // x is an operator
            pop from the stack the correct number of operands for the operator;
            perform the operation x and store the result (if any) onto the stack;
        }
    }
}
```



Two Essential Algorithms

- Combining two algorithms enables computers to handle human-written expressions
 - Infix-to-Postfix conversion
 - Postfix evaluation (just mentioned)





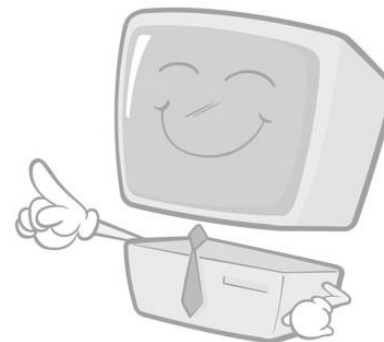
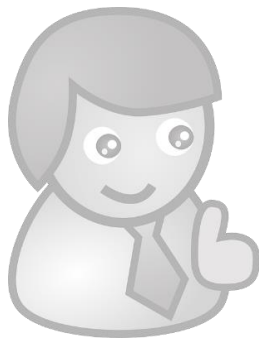
Infix to Postfix Conversion

- Observations
 - Number of operands and operators do not change
 - Order of operands (A, B, C...) do not change

Infix
 $A/B-C+D*E-A*C$

Conversion
algorithm

Postfix
 $AB/C-DE*+AC*-$





Infix to Postfix Conversion

- Method 1

- Fully **parenthesize** the expression (based on the operator priorities)
- **Move all operators** so that they replace their corresponding right parentheses
- **Delete all parentheses**

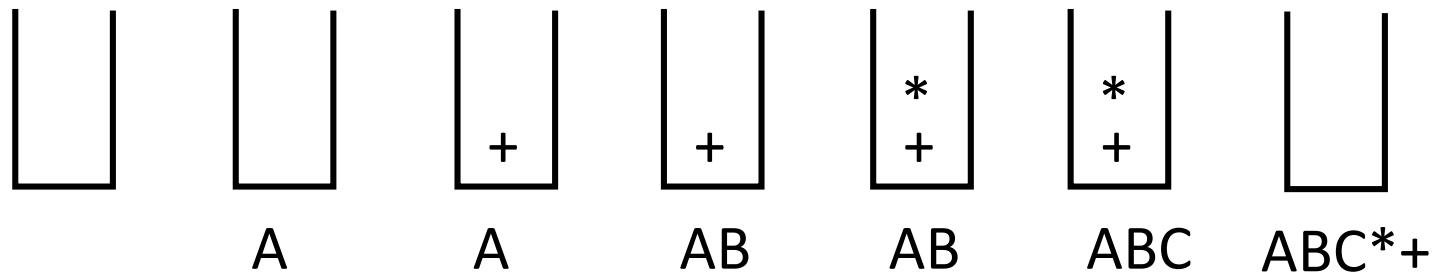
$$A/B-C+D*E-A*C \quad \rightarrow \quad ((((A/B)-C)+(D*E))-(A*C))$$

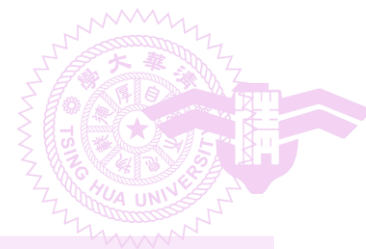
$$((((AB/C-(DE* +(AC* - \quad \rightarrow \quad AB/C-DE*+AC* -$$



Infix to Postfix Conversion

- Stack-based algorithm
 - Create a stack
 - Scan the input infix expression left to right
 - Bypass each incoming operand to the output
 - For each incoming operator
 - First, continuously pop from the stack an operator (the top) if the top has equal or lower priority than the incoming operator
 - Then, push the incoming operator onto the stack
 - Pop all operators upon the end of an expression
- Example: $A + B * C$





Parentheses Handling

- We want the stack algorithm to handle parentheses similarly to handling operators
- Specialized rules for left parenthesis
 - Incoming left parenthesis has the highest priority (i.e., always gets pushed onto the stack)
 - In-coming priority (ICP) = 0
 - Only gets popped from the stack upon a matched right parenthesis
 - Otherwise, behaves as one with the lowest priority
 - In-stack priority (ISP) = 8

Priority	Operator
0	In-coming (
1	Unary minus (負號), !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	=, !=
6	&&
7	
8	In-stack (



Example

- $A*(B+C)/D$

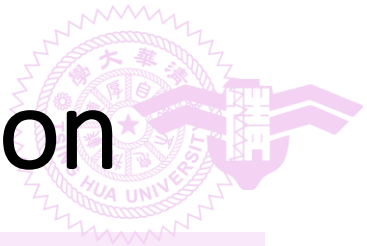
Incoming token	Stack	Output	Note
Empty	Empty	Empty	
A			
*			
(
B			
+			
C			
)			
/			
D			
Done			



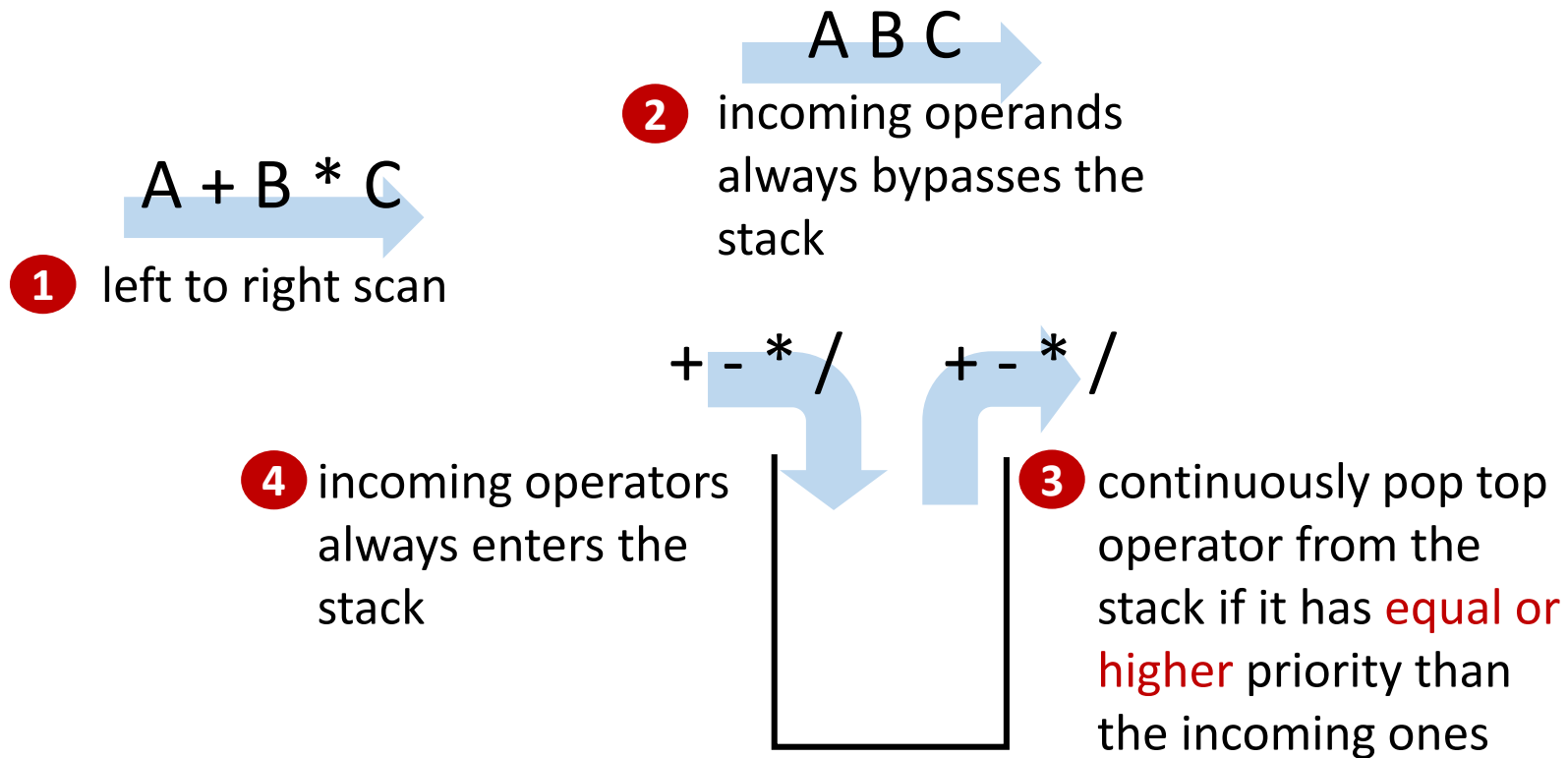
Example

- $A*(B+C)/D$

Incoming token	Stack	Output	Note
Empty	Empty	Empty	
A	Empty	A	Bypass operands
*	*		
(*(ICP('(') higher than ISP('*')
B	*(AB	Bypass operands
+	*(+		ICP('+') higher than ISP('(')
C	*(+	ABC	Bypass operands
)	*	ABC+	Pop until a left parenthesis
/	/	ABC+*	ICP('/') == ISP('*')
D	/	ABC+*D	Bypass operands
Done	Empty	ABC+*D/	Pop all operators



Recap Infix to Postfix Conversion





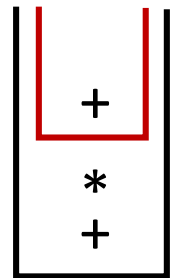
Recap Parenthesis Handling

- Incoming left parenthesis has the highest priority
 - It always enters the stack without popping any stacked operator
- In-stack left parenthesis has the lowest priority
 - It never gets popped from the stack until the right parenthesis appears

$$A+B*(C+D)$$

- Different perspective ¹

- Left parenthesis creates an isolated, nested stack
- Right parenthesis cleans up a nested stack



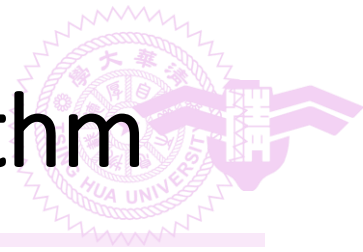
1. Contributed by Mr. 陳德暉 (101061132) on April 2, 2015



Infix to Postfix Algorithm

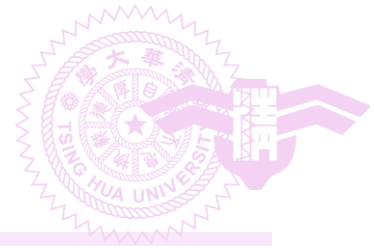
```
void Postfix(Expression e)
{
    Stack<Token>stack; // initialize the stack
    stack.Push('#');
    for (Token x = NextToken(e); x != '#'; x = NextToken(e))
        if (x is an operand) cout << x;
        else if (x == ')') { // pop until a left parenthesis
            for (;stack.Top( ) != '('; stack.Pop( ))
                cout << stack.Top( );
            stack.Pop( ); // remove the left parenthesis
        } else { // x is a operator
            for (; isp(stack.Top( )) <= icp(x); stack.Pop( ))
                cout << stack.Top( ); // higher or equal priority

            stack.Push(x);
        }
    // end of expression; empty the stack
    for ( ; !stack.IsEmpty( ); cout << stack.Top( ), stack.Pop( ));
    cout << endl;
}
```



Limitations of the Current Algorithm

- Characters to tokens conversion (**parser**)
 - Energy = Mass * LightSpeed * LightSpeed
 - Area = 3.14 * radius1 * radius2
- **Grammar**
 - $X = A - B + -A$
computers need rules to differentiate the two minus symbols; Otherwise, the aforementioned postfix algorithm cannot work correctly.
- More techniques are available in a compiler course



Outline

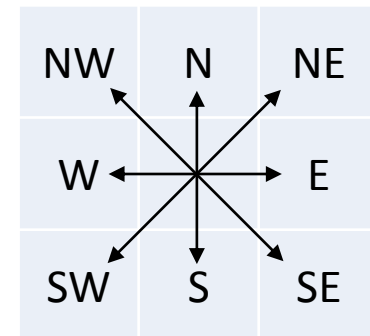
- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- 3.6 Evaluation of expressions
- **3.5 A mazing problem**



A Mazing Problem

- 2D array maze representation
 - '1' implies a blocked direction
 - '0' means otherwise
 - Borders are surrounded by '1'
- Allowable moves
 - Non-blocked squares of the eight neighboring squares
- How can a program get through the maze?

1	1	1	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	0	1	1
1	1	0	1	0	0	0	1	0	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	0	0	0	1	0	1	1	1
1	0	1	1	0	1	1	1	0	1	1
1	1	0	1	0	1	1	1	0	1	1
1	0	0	1	1	0	0	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1





Conceptual Algorithm

- Steps
 1. Push current coordinates and direction onto a stack
 2. Find a new and valid move
 - Starting from the north and looking clockwise
 - Retract from a dead-end using the information of the stack
 3. Use an array to mark the visited positions



Algorithm (Pseudo Code)

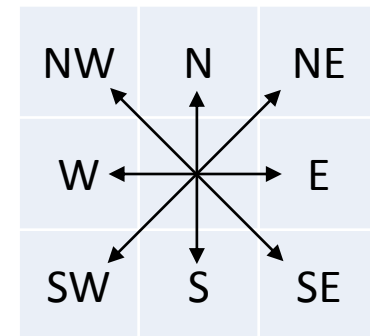
```
struct Offsets
{
    int di, dj;
}

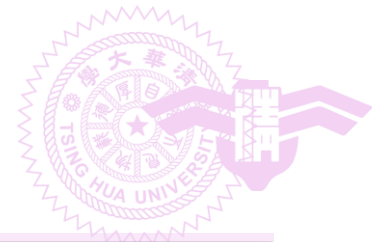
enum directions {N, NE, E,
SE, S, SW, W, NW};

Offsets move[8];

struct Items
{
    int x, y, dir;
}
```

q	move[q].di	move[q].dj
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1





Algorithm (Pseudo Code)

```
initialize a stack
push the starting coordinates and direction onto the stack
while (the stack is not empty) {
    (i, j, dir) = the last element of the stack;
    remove the last element of the stack;
    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 new move
            mark [g][h] = 1;
            dir = next direction to try;
            push (i, j, dir) onto the stack;
            (i, j, dir) = (g, h, N) ;
        }
    }
}
cout << "No path in maze." << endl;
```

- Each position can be visited at most once.
- At most eight valid moves from each position
- $O(\text{size of the array})$ time



Stack Provided by C++ Library

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> s;
    for(int i=0; i < 5; i++){
        s.push(i);
    }
    while(!s.empty())
    {
        cout << s.size() << " ";
        cout << s.top() << endl;
        s.pop();
    }
}
```

output

```
5 4
4 3
3 2
2 1
1 0
```

Reference of STL's Stack

<http://en.cppreference.com/w/cpp/container/stack>