# Chapter 3 Stacks and Queues

C++ Template

Stack ADT

Queue ADT

Subtyping and Inherence in C++

# Templates in C++

- Template function in C++ makes it easier to reuse classes and functions.
- A template can be viewed as a variable that can be instantiated to any data type, irrespective of whether this data type is a fundamental C++ type or a user-defined type.
  - fundamental C++ type: int, float, …
  - User-defined type: Rectangle, Polynomial, …

# Selection Sort Using Template

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

# Selection Sort Using Template (contd.)

```
float farray[100];
int intarray[250];
…
// Assume that the arrays are initialized at this point
SelectSort(farray, 100);
SelectSort(intarray, 250);
```

# Change Size of 1-D array

```
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 ];                       // new array
        int number = min ( oldSize, newSize );             // number to copy
        cop y (a, a + numbe, temp);
        delete [] a;                                       //release old memory
        a = temp;
}
```

If *T* is a user defined class, operators <, = should be overloaded (重載).

# Using Templates to Represent Container Classes

- A container class is a class that represents a data structure that contains or stores a number of data objects.

# Using Templates to Represent Container Classes

Container *Bag* for Integer

```
class Bag
{
public:
        Bag ( int bagCapacity = 10 );          // 建構子
        ~Bag( );                               // 解構子

        int Size( ) const;                     // 回傳袋中的元素個數
        bool IsEmpty( ) const;                 // 如果袋子是空的就回傳 true；不然就回傳 false
        int Element( ) const;                  // 回傳袋子中的一個元素

        void Push(const int);                  // 插入一個整數到袋子裡
        void Pop( )                            // 從袋子裡刪除一個整數

private:
        int *array;
        int capacity;                          // 陣列的容量
        int top;                               // 頂端元素在陣列裡的位置
};
```

```
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];
}

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

void Bag::Pop ( ) {
        if ( IsEmpty ( ) ) throw “Bag is empty, cannot delete”;
        int deletePos = top / 2;
        copy (array + deletePos + 1, array + top + 1, array + deletePos);
                        //  使陣列緊湊
        top - -;
}
```

# Using Templates to Represent Container Classes

```
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;
        int top;
}
```

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

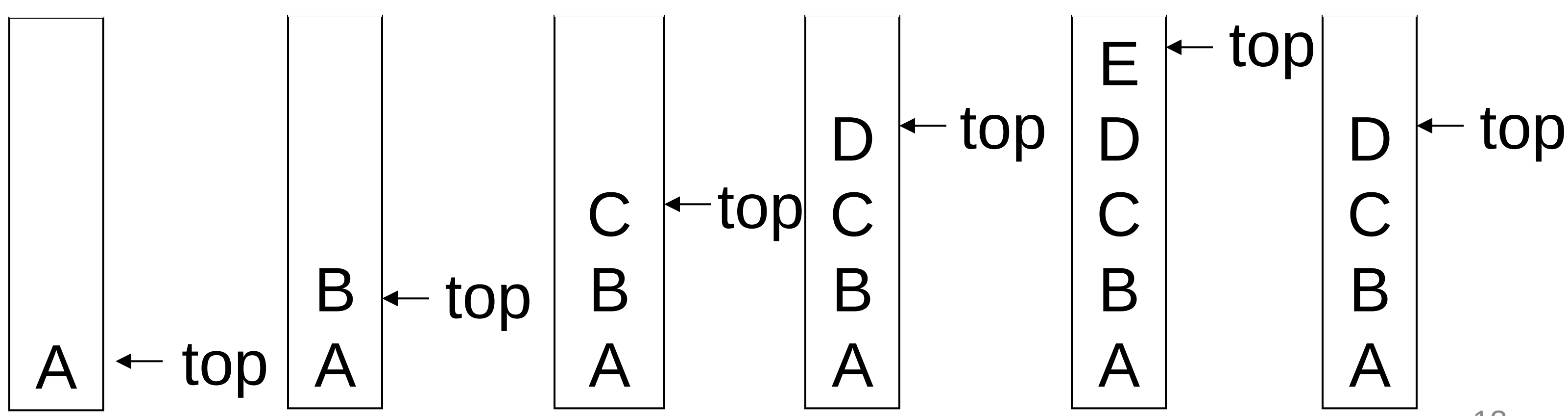
```
template < class T >
Bag < T >::Bag (int bagCapacity) : capacity (bagCapacity) {
        if (capacity < 1) throw "Capacity must be > 0";
        array = new T [capacity];
        top = -1;
}

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 is empty, cannot delete";
        int deletePos = top / 2;
        copy (array + deletePos + 1, ayyay + top + 1, array + deletePos);
                        //  使陣列緊湊
        array [ top-- ].~T ( ) ; // T 的解構子
}
```

# Stack ADT

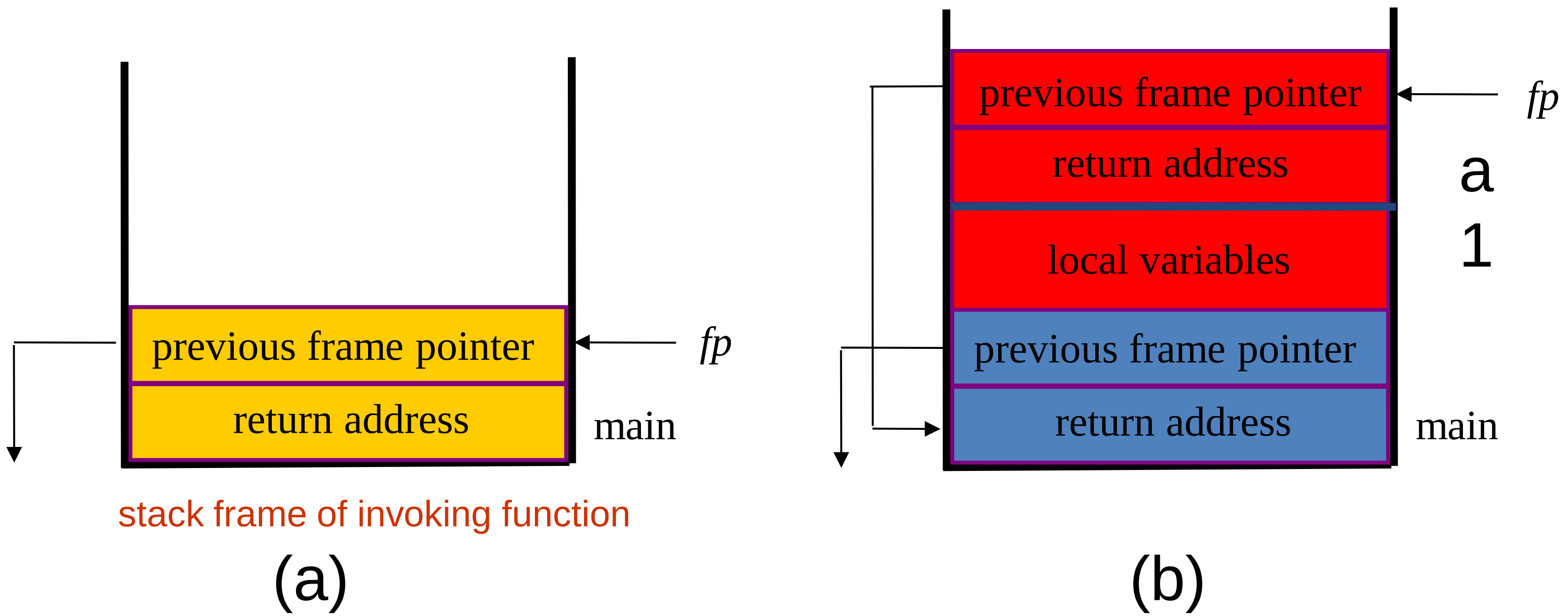
- Ordered list $A = a_0, a_1, a_2, \ldots, a_{n-1}$, where $a_i$ is called atom or element
  - The null or empty list, denoted by (), has $n = 0$ elements.
- Stack: Last-In-First-Out(LIFO) list
- A stack is an ordered list in which insertions and deletions are made at one end called the top.
- Given a stack S = $(a_0, a_1, a_2, \ldots, a_{n-1})$
  - $a_0$ is bottom
  - $a_{n-1}$ is top
  - $a_i$ is on top of $a_{i-1}$, $0 < i < n$.

# Stack: Last-In-First-Out (LIFO) List

- Insert (Push)
  - Add an element into a stack
- Delete (Pop)
  - Get and delete an element from a stack



# An Application of Stack: Stack Frame of Function Call



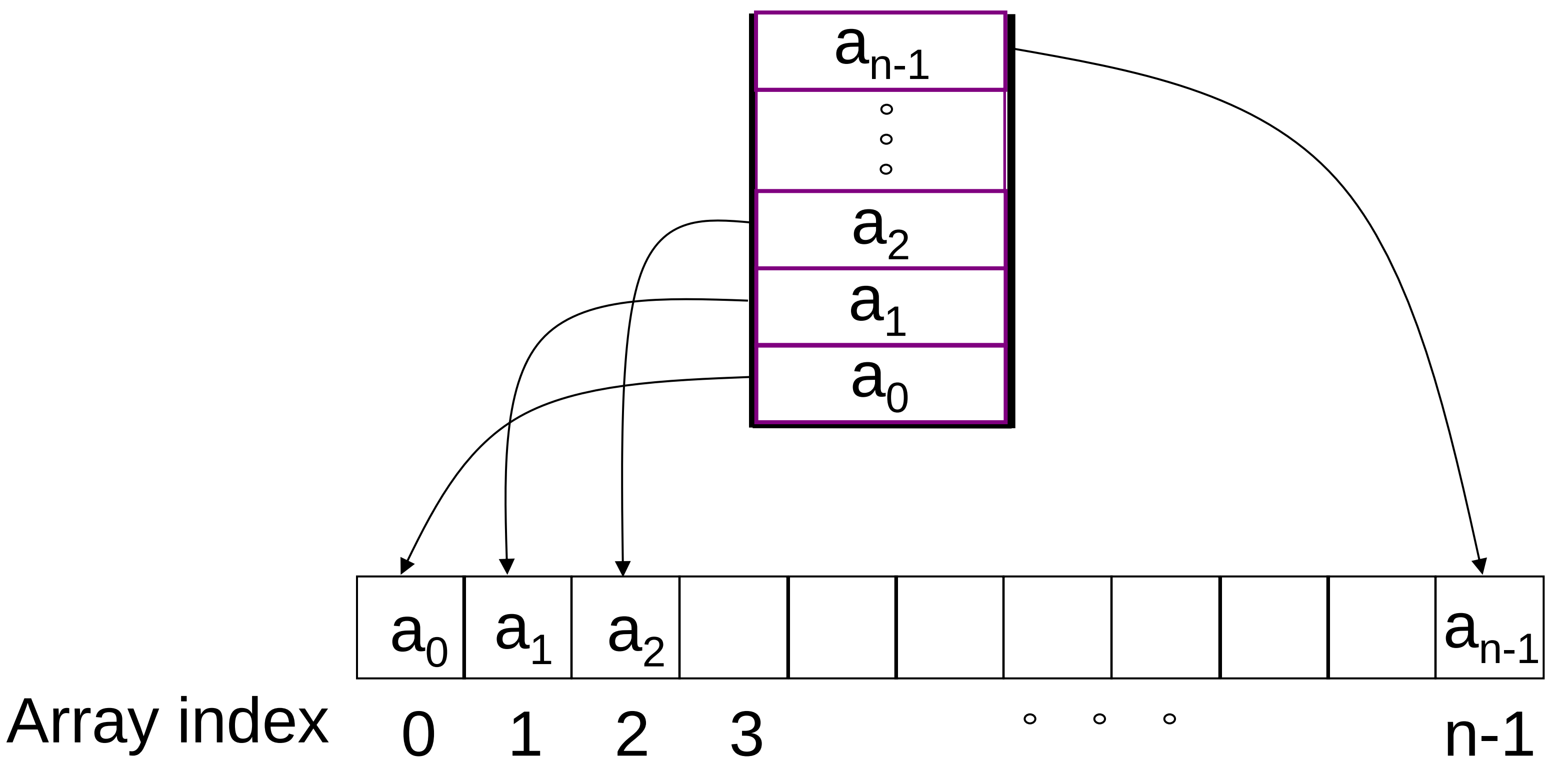

System stack after function call

# Stack ADT

```
template < class T >
class Stack
{ // a finite ordered list with zero or more elements
public:
    Stack (int stackCapacity = 10);
    // create an empty stack whose initial capacity is stackCapacity
    bool IsEmpty ( ) const;
    // if no 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 the top of the stack
    void Pop ( );
    // delete the top element of the stack
};
```

# Implementation of Stack by Array

- How to check whether a stack is full or empty?



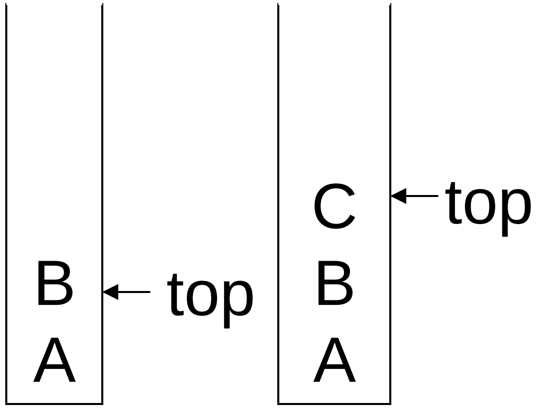
# Implementation with Template

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

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 is empty";
            return stack[top];
}
```

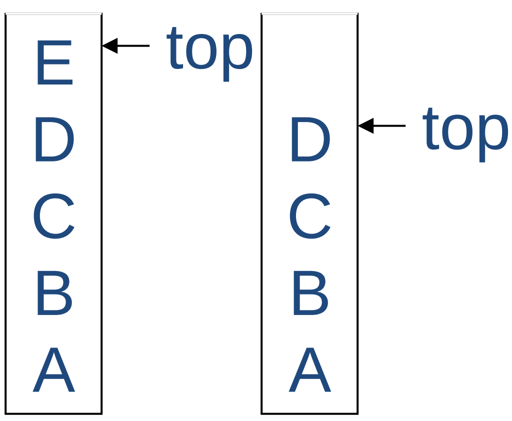
編譯器 (compiler) 會將 inline 函數的部份最佳化，通常會把 inline 函數的程式直接插入執行檔編譯，避免過多的函數呼叫

# Implementation with Template

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

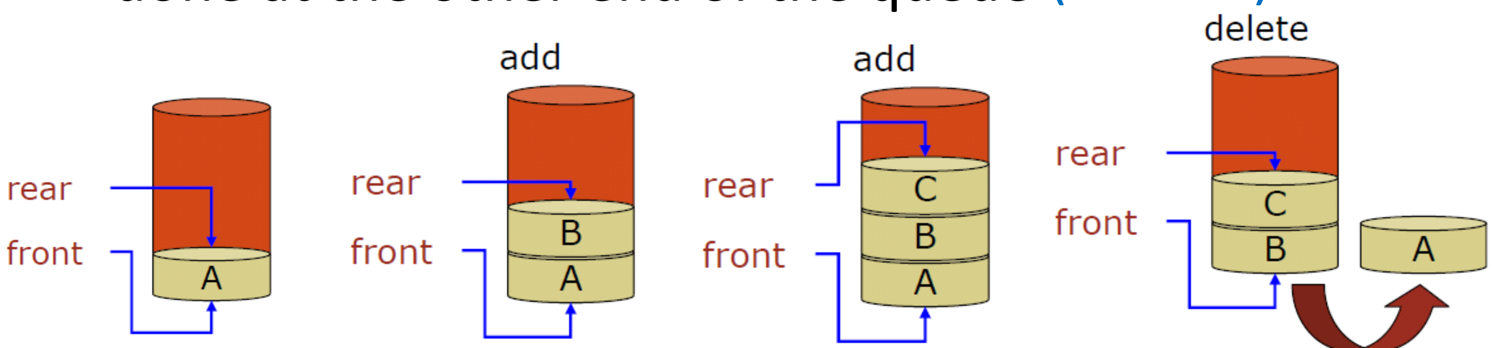



```
template < class T >
void Stack < T >::Pop ( )
{ // delete top element from the stack
      if (IsEmpty ( ) ) throw “Stack is empty. Cannot delete.”;
      stack [ top-- ].~T();     // destructor for T
}
```



# Queues

- Queue is an ordered list in which insertions take place at one end and all deletions take place at the opposite end.
- The first element inserted into a queue is the first element removed. (First-In-First-Out list, FIFO)
  - Insertion or enqueue: Placing an item in a queue is done at the end of the queue ("rear")
  - Deletion or dequeue: Removing an item from a queue done at the other end of the queue ("front")

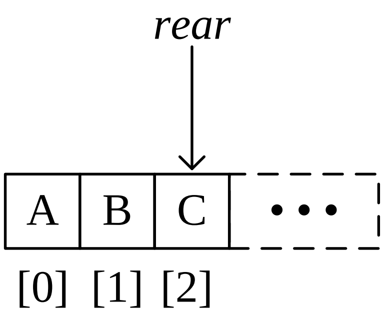


# Queue ADT

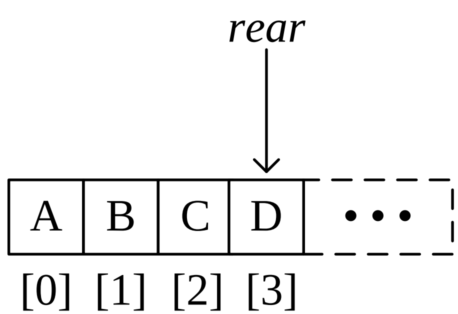
```
template < class T >
class Queue
{ // a finite ordered list with zero or more elements.
public:
    Queue (int queueCapacity = 0);
    // create an tmpty queue whose initia capacity is stackCapacity
    bool IsEmpty ( ) const;
    // if no of elements in the queue is 0, return true else return false
    T& Front ( ) const;
    // return the element at the front of the queue
    T& Rear ( ) const;
    // return the element at the rear of the queue
    void Push (const T& item);
    // insert item at the rear of the queue
    void Pop ( );
    // delete the front element of the queue
} ;
```

# Implementation of Queue

- queue[0] be the front element
  - Add an element into a queue



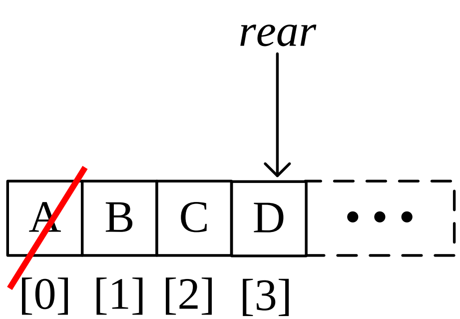

(a)



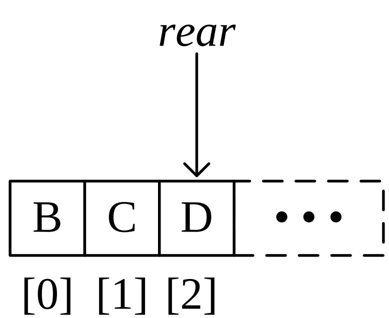

(b)

$\Theta(1)$

  - Get and delete an element from a queue




(a)

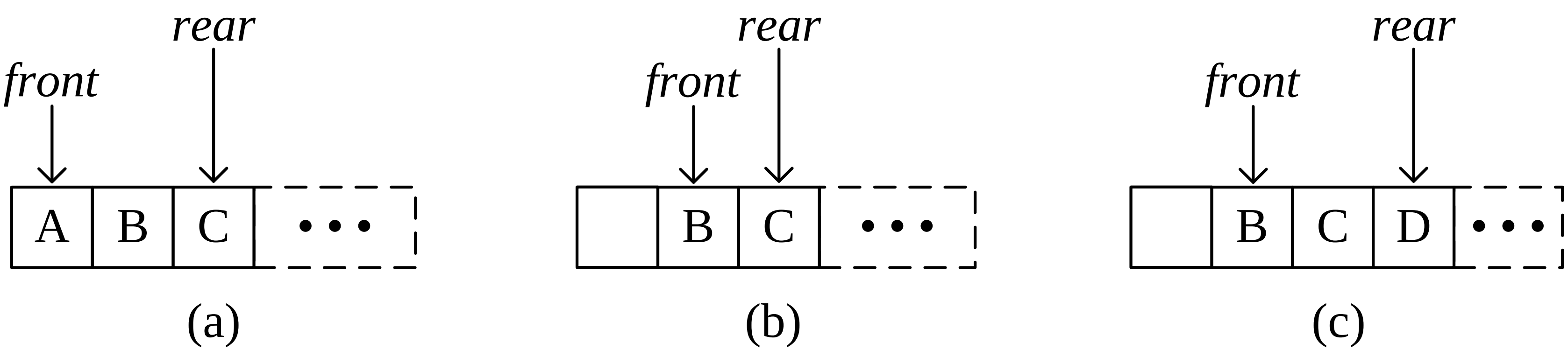



(b)

$\Theta(n)$

# Sequential Queue

- *front* keeps track of the front element
- *rear* keeps track of the end element



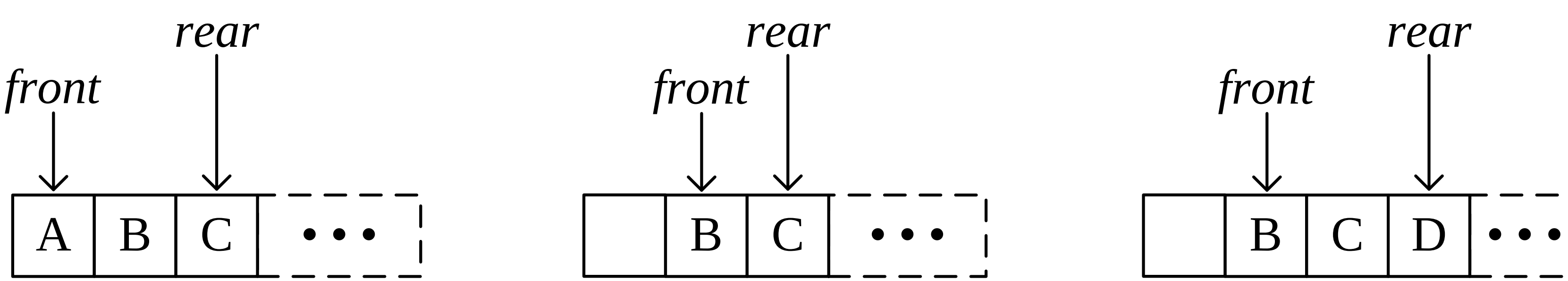
# Application: Job Scheduling

| front | rear | Q[0] Q[1] Q[2] Q[3] | Comments |
|---|---|---|---|
| 0 | 0 | | Queue is empty |
| 0 | 1 | J1 | J1 is added |
| 0 | 2 | J1 J2 | J2 is added |
| 0 | 3 | J1 J2 J3 | J3 is added |
| 1 | 3 | J2 J3 | J1 is deleted |
| 2 | 3 | J3 | J2 is deleted |

Insertion and deletion from a sequential queue

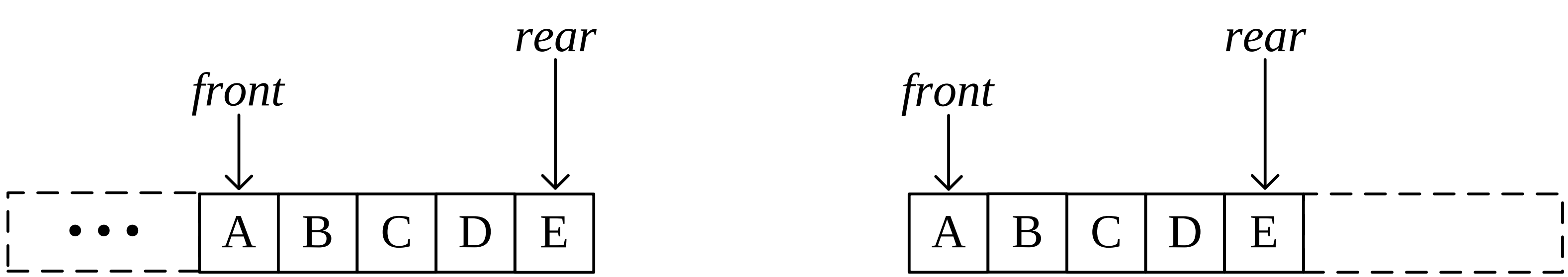
# Sequential Queue

- *front* keeps track of the front element
- *rear* keeps track of the end element



(a) (b) (c)

- Problem: if *rear* == *capacity* -1 and *front* >0



(a) Before shift (b) After shift

# Implementation 2: Regard an Array as a Circular Queue

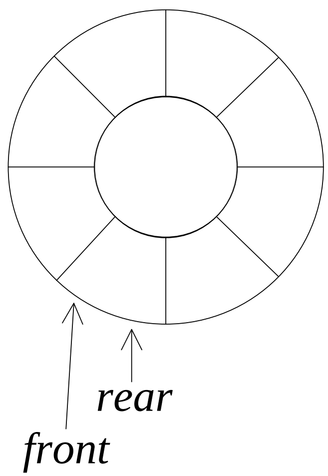
- Two indices
  - front: one position counterclockwise from the first element
  - rear: current end
- Problem
  - In order to distinguish whether a circular queue is full or empty, one space is left when queue is full

Rule:

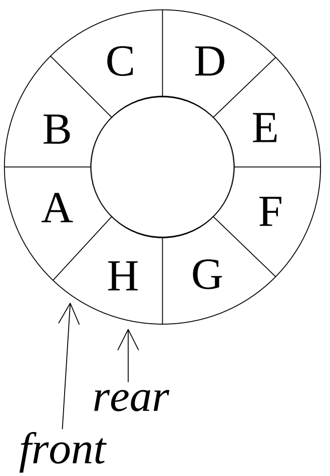
Next to $capacity - 1$ is $0$
Precede $0$ is $capacity - 1$

# Circular Queue

- Problem: when *front* == *rear* ，empty or full queue cannot be distinguished




(a) 圖3.8(a)的佇列連續執行了三次的移除動作後的結果



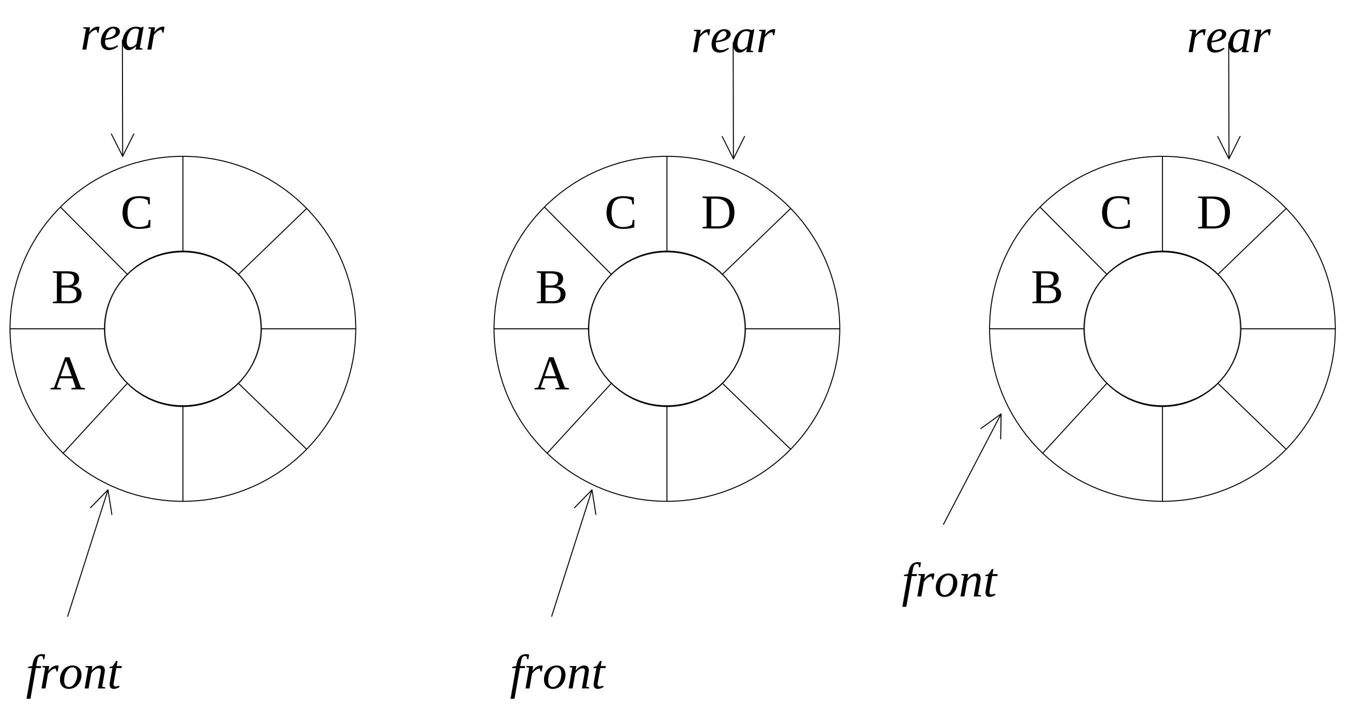

(b) 圖3.8(a)的佇列連續做了五次的加入動作後的結果

# Circular Queue

- Insertion: if (*rear* == *capcaity*-1) *rear* = 0; else *rear*++;

||

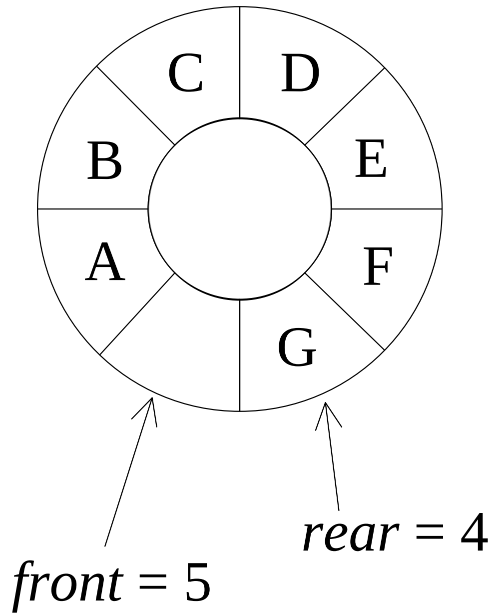
*rear* = (*rear*+1)%*capacity*




(a) initial (b) after addition (c) after deletion

# Implementation of Circular Queue

```
template <class T>
void Queue<T>::Push(const& x);
{ // inert x to rear of the queue
        if ((rear + 1) % capacity = = front)
        { // queue full, double capacity
                // add your code here
        }

        rear = (rear + 1) % capcity;
        queue [rear ] = x;
}
```




(a) Circular queue full

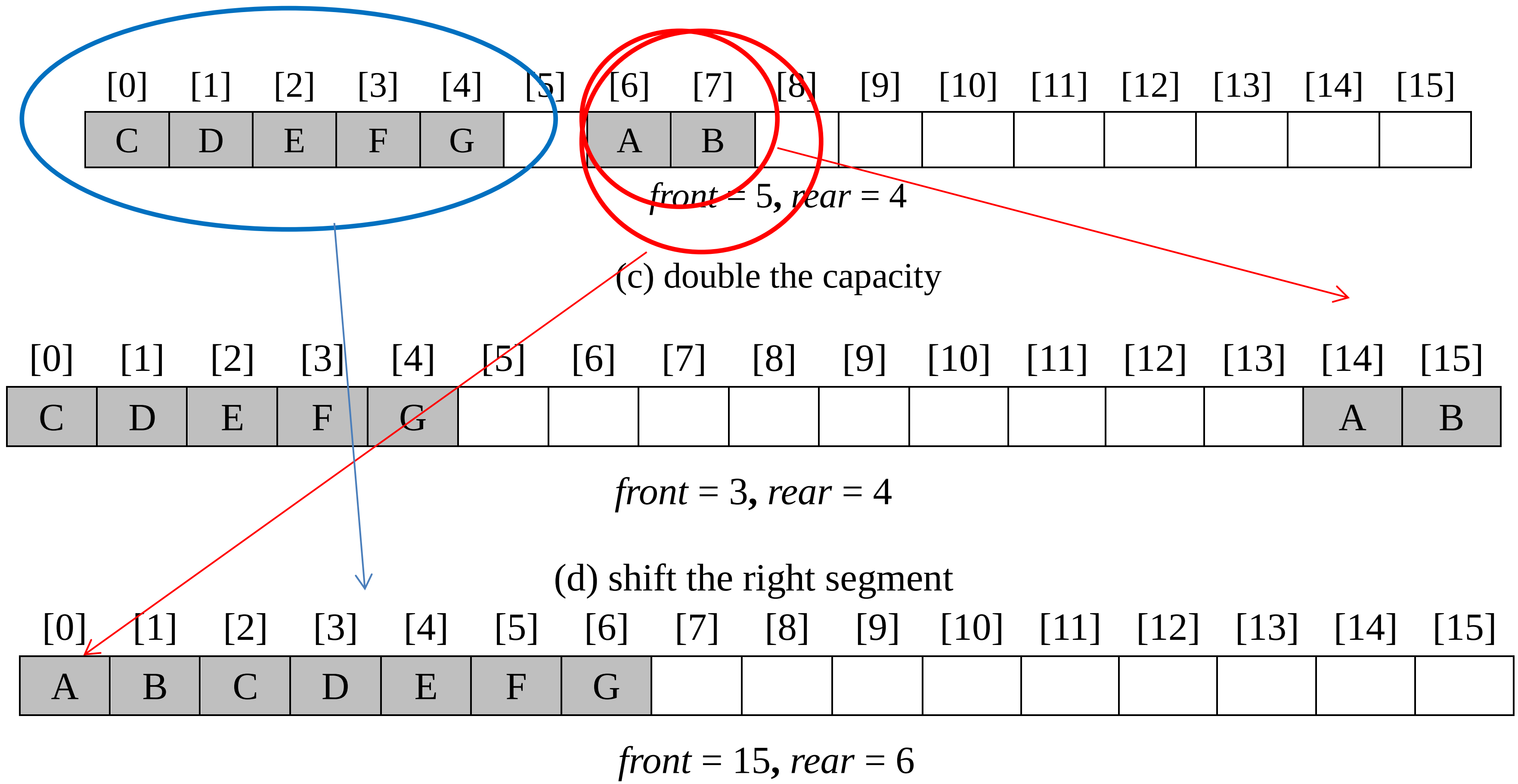
| *queue* | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|---|---|---|---|---|---|---|---|---|
| | C | D | E | F | G | | A | B |

*front* = 5**,** *rear* = 4

(b) flattened view of circular full queue

# Implementation of Circular Queue

- An efficient implementation when doubling capacity




(c) double the capacity

(d) shift the right segment

(e) alternative configuration

# Doubling Queue Capacity

```
// 配置一個雙倍容量的陣列
T* newQueue = new T [2*capacity];

// 從 queue 複製至 newQueue
int start = (front + 1) % capacity;
if (start < 2)
        // 沒捲繞
        copy (queue + start, queue + start + capacity − 1, newQueue);
else
{ // 佇列捲繞
        copy (queue + start, queue + capacity, newQueue);
        copy (queue, queue + rear + 1, newQueue + capcity − start);
}

// 切換至 newQueue
front = 2 * capacity – 1;
rear = capacity – 2;
capacity *= 2;
delete [] queue;
queue = newQueue;
```

# Delete from a Circular Queue (Pop)

```
template <class T>
void Queue<T>::Pop( );
{ //  刪除佇列前端的元素
        if (IsEmpty( )) throw "Queue is empty. Cannot delete.";
        front = (front + 1) % capacity;
        queue [front ].~T(); // T 的解構子
}
```

# Subtyping and Inheritance in C++

- Inheritance is used to express subtype relationships between two ADTs.
  - A parent-child relationship between two ADTs (classes)
- If B inherits from A, then B IS-A A. Also, A is more general than B.
  - VW Beetle IS-A Car
  - Eagle IS-A Bird
- Inheritance allows sharing of the behavior of the parent class into its child classes
- one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance

# Inheritance

- A derived class (subclass,  child class) inherits all the non-private members of the base class (superclass, parent class)
  - data ( local variable)
  - functions
- Inherit (extend, derive) means becoming a subclass of another class
- Inherited members from public inheritance have the same level of access in the derived class as they did in the base class.
- The derived class can reuse the implementation of a function in the base class or implement its own function, with the exception of constructor and destructor.
  - Child class can add new behavior or override existing behavior from parent

# Derived Class-Public Inheritance

```
class Derived : public Base {
  // Any members that are not listed are inherited unchanged except
  // for constructor, destructor, copy constructor, and operator=
 public:
  // Constructors, and destructors if defaults are not good
  // Base members whose definitions are to change in Derived
  // Additional public member functions
 private:
 // Additional data members (generally private)
 // Additional private member functions
 // Base members that should be disabled in Derived
};
```

# Visibility rules

- Any private members in the base class are not accessible to the derived class
  - (because any member that is declared with private visibility is accessible only to methods of the class)
- What if we want the derived class to have access to the base class members?
  1) use public access =>public access allows access to other classes in addition to derived classes
  2) use a friend declaration=>this is also poor design and would require friend declaration for each derived class
  3) make members protected =>allows access only to derived classes

     A *protected class member* is private to every class except a derived class, but declaring data members as protected or public violates the spirit of encapsulation
  - write accessor and modifier methods =><u>the best alternative</u>

    Note: However, if a protected declaration allows you to avoid convoluted code, then it is not unreasonable to use it.

# Class Inheritance Example

```
class Bag
{
public:
        Bag ( int bagCapacity = 10 );              //  建構子
        ~Bag( );                                   //  解構子

        int Size( ) const;                         //  回傳袋中的元素個數
        bool IsEmpty( ) const;                     //  如果袋子是空的就回傳 true；不然就回傳 false
        int Element( ) const;                      //  回傳袋子中的一個元素

        void Push(const int);                      //  插入一個整數到袋子裡
        void Pop( )                                //  從袋子裡刪除一個整數

private:
        int *array;
        int capacity;                              //  陣列的容量
        int top;                                   //  頂端元素在陣列裡的位置
};
```

# Inherited Member Initialization

- Initialized in two ways:
  - If the base class has only a default constructor=> initialize the member values in the body of the derived class constructor
  - If the base class has a constructor with arguments=> the initialization list is used to pass arguments to the base class constructors

syntax (for Single Base Class)

DerivedClass ( derivedClass args ) : BaseClass ( baseClass args ) {

    DerivedClass constructor body

    }

- The set of derived class constructor arguments may contain initialization values for the base class arguments

# Class Derivation
# Constructors and Destructors

- Constructors and destructors are not inherited
  - Each derived class should define its constructors/destructor
  - If no constructor is written=> hidden constructor is generated and will call the base default constructor for the inherited portion and then apply the default initialization for any additional data members
- When a derived object is instantiated, memory is allocated for
  - Base object
  - Added parts
- Initialization occurs in two stages:
  - the base class constructors are invoked to initialize the base objects
  - the derived class constructor is used to complete the task
- The derived class constructor specifies appropriate base class constructor in the initialization list
  - If there is no constructor in base class, the compiler created default constructor used
- If the base class is derived, the procedure is applied recursively

# Class Inheritance Example(Cont.)

```
class Stack : public Bag {
    public:
        Stack(int MaxSize = 10);                // constructor
        ~Stack();                               // destructor
        int  Top() const;                       // return the top element from stack
        void  Pop();                            // delete the top element  from stack
};
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];
}
int Stack::Pop() const{
    if (IsEmpty()) throw “stack is empty.”;
    return array[top];
}
```

# Class Inheritance Example (Cont.)

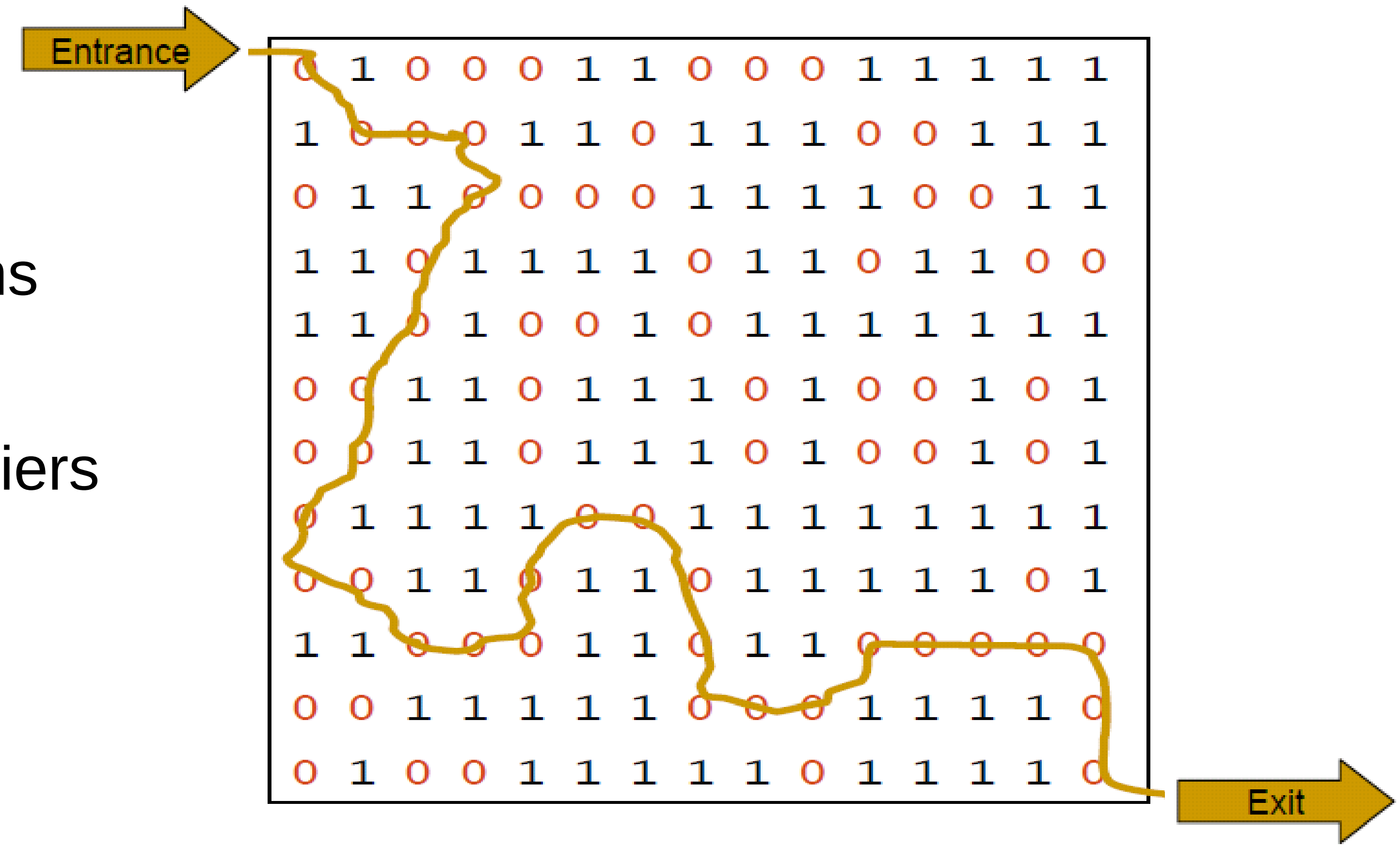
```
Bag b(3);              // uses Bag constructor to create array of size 3
Stack  s(3);           // uses Stack constructor to create array of size 3

b.Push(1); b.Push(2); b.Push(3);
                       // use Bag::Push
s.Push(1); s.Push(2); s.Push(3);
                       // use Bag::Push
                       // Stack::Push not defined, so use Bag::Push.

b.Pop();               // uses Bag::Pop, which calls Bag::IsEmpty
s.Pop();
// uses Stack::Pop, which calls Bag::IsEmtpy  because these  have not been
// redefined in Stack.
s.Size();               // use Bag::Size
s.Element();            // use Bag::Element
```

# A Mazing Problem

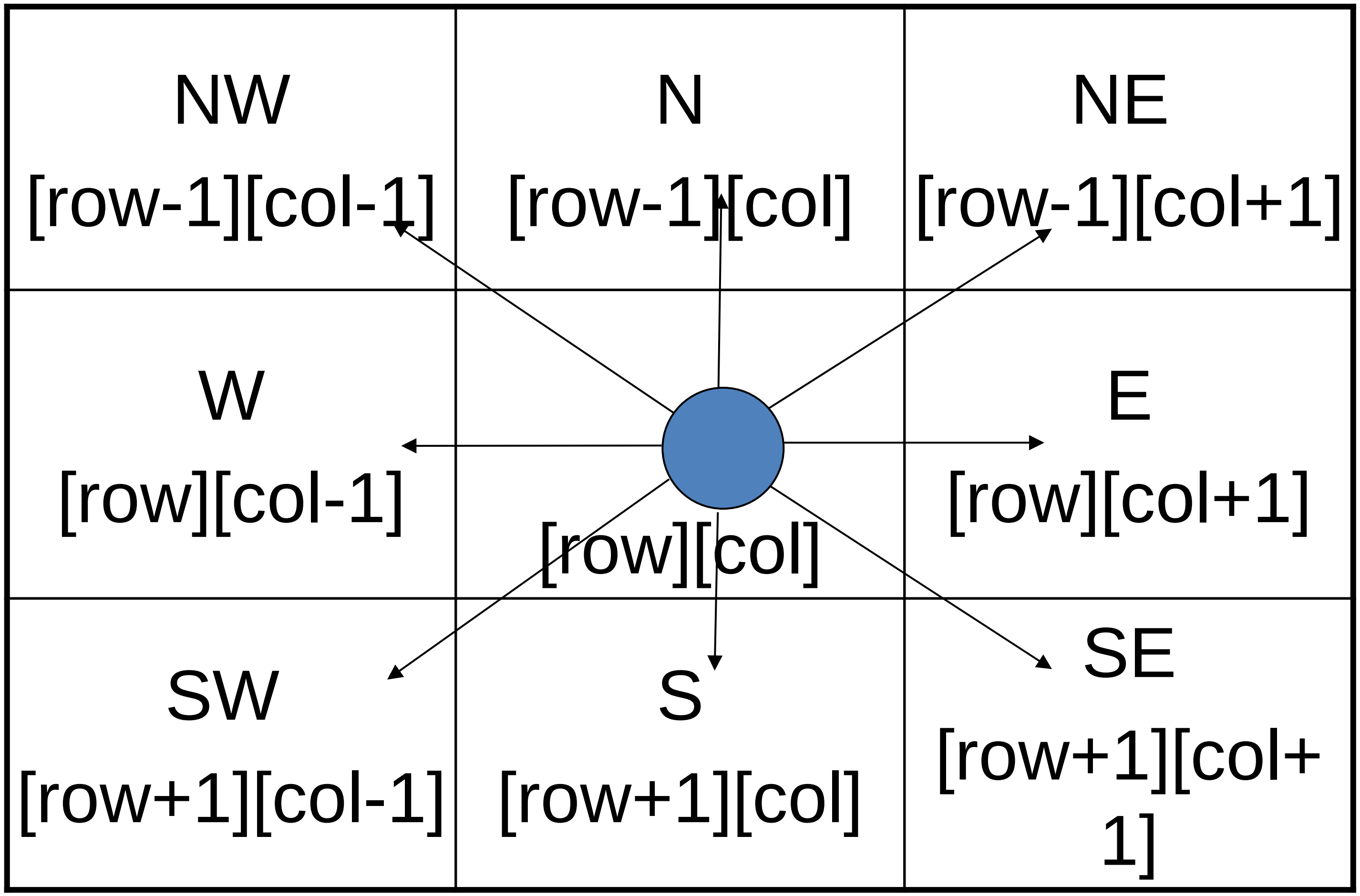
- Example: find the path

0: paths

1: barriers



# A Possible Representation



```
typedef struct {
  int a; /* row */
  int b; /* col */
} offsets;
offsets move[8];
/*array of moves for each direction*/
```

next_row = row + move[dir].a;
next_col = col + move[dir].b;

| Name | Dir | Move[dir].a | Move[dir].b |
|---|---|---|---|
| N | 0 | -1 | 0 |
| NE | 1 | -1 | 1 |
| E | 2 | 0 | 1 |
| SE | 3 | 1 | 1 |
| S | 4 | 1 | 0 |
| SW | 5 | 1 | -1 |
| W | 6 | 0 | -1 |
| NW | 7 | -1 | -1 |

# Initial Attempt at A Maze Traversal Algorithm

- A second two-dimensional array, **mark**, records the maze positions already checked.
- Use a stack to save current path and direction.
  - We can return to it and try another path if we take a hopeless path.
  - The stack size is at most *mxn*, which is the number of positions in the maze.

# Use a Stack to Keep Pass History

- What is the maximal size of the stack?
  - A maze is represented by a two-dimensional array *maze[m][p]*
  - Since each position is visited at most once, at most $m \times p$ elements can be placed in the stack

```
typedef struct {
  int x;
  int y;
  int dir;
} item;
item stack[m*p];
```

```
while (list 不是空的)
{
    (i, j, dir) = list 尾端的座標與方向;
    刪除 list 的最後一個元素;
    while (從位置 (i, j) 還有可以移動的方向)
    {
        (g, h) = 下一個移動的座標;
        if ((g = = m) && (h = = p)) 成功;
        if ((!maze [g][h])    // 合法移動
            && (!mark [g][h])) // 之前沒到過這裡
        {
            mark [g ][h] = 1;
            dir = 下一個要試的方向;
            加入 (i, j, dir) 到 list 的尾端;
            (i, j, dir) = (g, h, N) ;
        }
    }
}
cout << "No path in maze." << endl;
```

```
0 0 0 0
0 1 0 0
0 1 1 1
0 0 0 0
```

```
Stack              Action
         Position
---------------------------------
------- (1,1) Mark (1,1)

(1,1,2) (1,2) Mark (1,2)

(1,2,2) (1,3) Mark (1,3)
(1,1,2)

(1,3,2) (1,4) Mark (1,4)
(1,2,2)
(1,1,2)

(1,4,4) (2,4) Mark (2,4)
(1,3,2)
(1,2,2)
(1,1,2)

(2,4,6) (2,3) Mark (2,3)
(1,4,4)
(1,3,2)
(1,2,2)
(1,1,2)

Move back

(1,4,4) (2,4)
(1,3,2)
(1,2,2)
(1,1,2)

Move back

(1,3,2) (1,4)
(1,2,2)
(1,1,2)

...
```

```
void Path(const int m, const int p)
{ // 輸出迷宮的一個路徑（如果有的話）； maze[0][i] = maze[m+1][i] =
  // maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1 。

    mark[1][1] = 1; // 從 (1, 1) 開始
    Stack<Items> stack(m*p);
    Items temp(1, 1, E); // 設定 temp.x、temp.y、與 temp.dir
    Stack.Push(temp);
    while (!stack.IsEmpty( )){
    { // 堆疊不是空的
        temp = stack.Top( );
        stack.Pop( );    // 彈出
        int i = temp.x; int j = temp.y; int d = temp.dir;
        while (d < 8){    // 往前移動
            int g = i + move[d].a; int h = j + move[d].b;
            if ((g = = m) && (h = = p)) { // 抵達出口
                 // 輸出路徑
                cout << stack;
                cout << i << " " << j << endl; // 路徑上的上兩個方塊
                cout << m << " " << p << endl;
                return;
            }
            if ((!maze [g ][h]) && (!mark [g ][h])) { // 新位置
                mark[g][h] = 1;
                temp.x = i; temp.y = j; temp.dir = d+1;
                stack.Push(temp); // 加入堆疊
                i = g; j = h; d = N; // 移到 (g, h)
            else d++; // 試下一個方向
        }
    }
    cout << "No path in maze." << endl;
}
```

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

# Example

maze:

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

mark:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | 1 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | |

to mark visited positions

Stack

(1, 1, E)

< *i*, *j*, *nextDir*>

Initial State

# Example

maze:

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

mark:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | 1 | 0 | 1 | 1 | 1 | 0 | |
| | 0 | 1 | 0 | 1 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | |

Stack

(1, 5, SW)
(1, 4, E)
(1, 3, E)
(2, 2, NE)
(1, 1, E)

< i, j, nextDir >

(1, 1, E) → (1, 2) barrier
(1, 1, SE) → (2, 2) Walk

Move forward

Can walk? (not wall and not visited)

# Example

maze:

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

mark:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | 1 | 0 | 1 | 1 | 1 | 0 | |
| | 0 | 1 | 0 | 1 | 0 | 0 | |
| | 0 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | |

Stack

- (3, 5, W)
- (2, 4, SE)
- (1, 5, SW)
- (1, 4, E)
- (1, 3, E)
- (2, 2, NE)
- (1, 1, E)

Move backward

# Example

maze:

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

mark:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | 1 | 0 | 1 | 1 | 1 | 0 | |
| | 0 | 1 | 0 | 1 | 0 | 0 | |
| | 0 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 1 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | |

Stack

(5, 6, S)
(5, 5, E)
(5, 4, E)
(4, 3, SE)
(3, 4, SW)
(3, 5, W)
(2, 4, SE)
(1, 5, SW)
(1, 4, E)
(1, 3, E)
(2, 2, NE)
(1, 1, E)

Result

# Analysis

- Space complexity:
  - An extra 2D array *mark* is used: $O(mp)$
- Time complexity:
  - The inner while-loop executes at most eight times for each direction. Each iteration takes $O(1)$ time. Therefore, the inner loop totally takes $O(1)$ time.
  - The outer while loop executes until the stack is empty.
    - If the number of zeros in the maze is $z$, at most $z$ positions can be marked.
  - Since $z$ is bounded above by $mp$, the computing time is $O(mp)$.

# Expressions

- The representation and evaluation of expressions is of great interest to computer scientists.
- Example
  - A / B – C + D * E – A * C
  - ((rear+1==front) || ((rear==MAX_QUEUE_SIZE-1) && !front))
    - operators : ==, +, -, ||, &&, ! 運算子
    - operands: rear, front, MAX_QUEUE_SIZE 運算元
    - parentheses: (, ) 括號

# Expressions

- Why expressions are important?
  - For example,
    - 1: 9 + 3 * 5 = ?
      - Case 1a: 9 + 3 * 5 = 24
      - Case 1b: 9 + 3 * 5 = 60
    - 2: 9 – 3 – 2 = ?
      - Case 2a: 9 – 3 – 2 = 4
      - Case 2b: 9 – 3 – 2 = 8
  - The difference between case 1a & case 1b
    - **Precedence rule** (優先權法則)
  - The difference between case 2a & case 2b
    - **Associative rule** (關連性法則)
- Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators.

> - How to generate the machine instructions corresponding to a given expression? Precedence rule + Associative rule

# Expressions

- **Infix** (The standard way of writing expressions)
  - Each operator comes in-between the operands
  - 2+3
- **Postfix** (Compilers typically use postfix)
  - Each operator appears after its operands
  - 23+
- **Prefix**
  - Each operator appears before its operands
  - +23

# Evaluation of Expressions

- Evaluating postfix expressions is much simpler than the evaluation of infix expressions.
  - There are no parentheses and precedence to consider.
  - To evaluate an expression we make a single left-to-right scan of it.
  - We can evaluate an expression easily by using a stack.
- A two phase approach
  - 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

# Phase 1: Postfix Expression Evaluation

**user** **computer**

| Infix | Postfix |
|---|---|
| 2+3*4 | 234*+ |
| a*b+5 | ab*5+ |
| (1+2)*7 | 12+7* |
| a*b/c | ab*c/ |
| (a/(b-c+d))*(e-a)*c | abc-d+/ea-*c* |
| a/b-c+d*e-a*c | ab/c-de*ac*-+ |

Postfix & prefix: no parentheses, no precedence

# Phase 2: Postfix Expression Evaluation

expression e = 6 2 / 3 – 4 2 * +

| Token | Stack [0] | [1] | [2] | Top |
|---|---|---|---|---|
| 6 | 6 | | | 0 |
| 2 | 6 | 2 | | 1 |
| / | 3 | | | 0 |
| 3 | 3 | 3 | | 1 |
| - | 0 | | | 0 |
| 4 | 0 | 4 | | 1 |
| 2 | 0 | 4 | 2 | 2 |
| * | 0 | 8 | | 1 |
| + | 8 | | | 0 |

| Token | Stack [0] | [1] | [2] | Top |
|---|---|---|---|---|
| 6 | 6 | | | 0 |
| 2 | 6 | 2 | | 1 |
| / | 6/2 | | | 0 |
| 3 | 6/2 | 3 | | 1 |
| – | 6/2–3 | | | 0 |
| 4 | 6/2–3 | 4 | | 1 |
| 2 | 6/2–3 | 4 | 2 | 2 |
| * | 6/2–3 | 4*2 | | 1 |
| + | 6/2–3+4*2 | | | 0 |

# Phase 2: Eval()

```
void Eval(運算式 e)
{// 計算運算式 e。假設在 e 裡的最後一個符號是'#'（一個符號可以是
 // 運算元、運算子、或是'#' ）。函式 NextToken 是用來從 e 中擷取
 // 下一個符號。這個函式用了一個堆疊 stack
    Stack<Token>stack; // 初始化 stack
    for (Token x = NextToken(e); x!= '#'; x=NextToken(e))
        if (x 是運算元) stak.Push(x) // 推入至 stack
        else {// 運算子
            為運算子 x 從 stack 中彈出正確數量的運算元 ；
            執行運算子 x 的運算並且將結果（如果有的話）推入到堆疊中 ；
        }
}
```

# Analysis of Eval()

- Suppose the input expression has length of *n*.
  - Space complexity: $O(n)$.
    - The stack used to buffer operands at most requires $O(n)$ elements.
  - Time complexity:
    - The function make only a left-to-right pass across the input.
    - The time spent on each operand is $O(1)$.
    - The time spent on each operator to perform the operation is $O(1)$.
    - Thus, the time complexity of Eval() is $O(n)$.

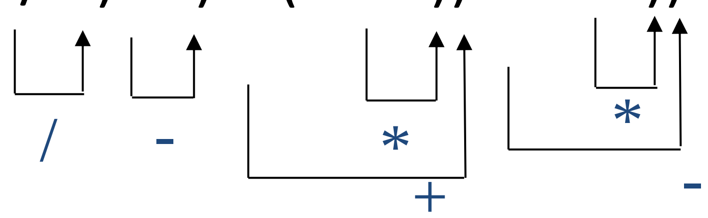
# Phase 1: Infix to Postfix Conversion

- Algorithm 1: Intuitive Algorithm
  1. Fully parenthesize expression

     a / b - c + d * e - a * c →((((a / b) - c) + (d * e)) – (a * c))

  2. All operators replace their corresponding right parentheses.

     ((((a / b) - c) + (d * e)) - a * c)) →((((a b / c – (d e * +a c * –




  3. Delete all parentheses.

     ab/c-de*+ac*-

# Phase 1: Infix to Postfix Conversion

- Algorithm 2
  - Scan string from left to right
  - Operands are taken out immediately
  - Operators are taken out of the stack as long as their in-stack precedence (isp) is higher than or equal to the incoming precedence (icp) of the new operator

$$isp(y) \geq icp(x)$$

  - If token == right parenthesis ")", unstack tokens until we reach the corresponding left parenthesis.

# Rules

- "(" has lowest in-stack precedence and highest incoming precedence
  - $isp(\text{"("}) = 8$
  - $icp(\text{"("}) = 0$
- No operator other than the matching right parenthesis ")" should cause it to get unstacked

| Priority | Operator |
|---|---|
| 1 | Unary minus, ! |
| 2 | *,/,% |
| 3 | +,- |
| 4 | <,<=,>=,> |
| 5 | ==,!= |
| 6 | && |
| 7 | \|\| |

# Example 1

e = a + b * c

| Token | Stack<br>[0][1][2] | Top | Output |
|---|---|---|---|
| a | | -1 | a |
| + | + | 0 | a |
| b | + | 0 | ab |
| * | + * | 1 | ab |
| c | + * | 1 | abc |
| # | | -1 | abc*+ |

# Example 2

e = a * b + c

| Token | Stack [0][1][2] | Top | Output |
|---|---|---|---|
| a | | -1 | a |
| * | * | 0 | a |
| b | * | 0 | ab |
| + | + | 1 | ab* |
| c | + | 1 | ab*c |
| # | | -1 | ab*c+ |

# Example 3

$e = a *_1 (b + c) *_2 d$

| Token | Stack [0][1][2] | Top | Output |
|---|---|---|---|
| a | | -1 | a |
| $*_1$ | $*_1$ | 0 | a |
| ( | $*_1$ ( | 1 | a |
| b | $*_1$ ( | 1 | ab |
| + | $*_1$ ( + | 2 | ab |
| c | $*_1$ ( + | 2 | abc |
| ) | $*_1$ | 0 | abc+ |
| $*_2$ | $*_2$ | 0 | abc+$*_1$ |
| d | $*_2$ | 0 | abc+$*_1$d |
| # | | -1 | abc+$*_1$d$*_2$ |

match (

$*_1 = *_2$

# Postfix

```
void Postfix(Expression e)
{// 把中置運算式 e 轉成後置運算式並輸出。NextToken 就跟函式 Eval（程式 3.18）裡
 // 的一樣。假設在 e 裡的最後一個符號是'#'。另外，'#'也用在堆疊的底部。
    Stack<Token>stack; // 初始化 stack
    stack.Push('#');
    for (Token x = NextToken(e); x != '#'; x = NextToken(e))
        if (x 是一個運算元) cout << x;
        else if (x = =')')
            {// 從堆疊中彈出直到出現'('
                for (;stack.Top( ) != '('; stack.Pop( ))
                    cout << stack.Top( );
                stack.Pop( ); // 從堆疊中彈出 '('
            }
        else { // x 是一個運算子
            for (; isp(stack.Top( )) <= icp(x); stack.Pop( ))
                cout << stack.Top( );
            stack.Push(x);
        }

    // 已經到了一個運算式的結尾；清空堆疊
    for (; !stack.IsEmpty( ); cout << stack.Top( ), stack.Pop( ));
    cout << endl;
}
```