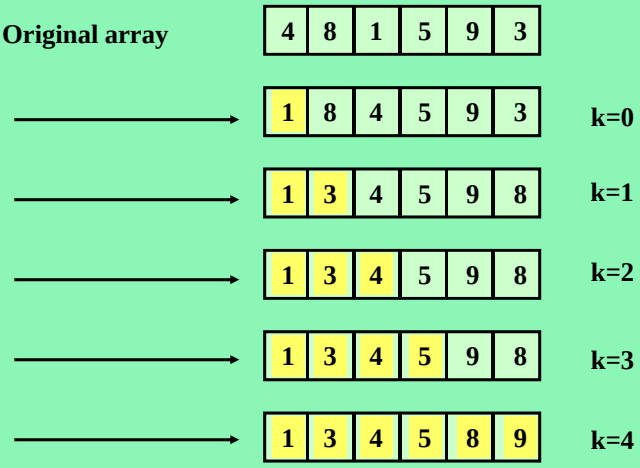
國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 3
Stacks and Queues

Outline

- **Templates in C++**
- **The Stack Abstract Data Type**
- **The Queue Abstract Data Type**
- **SubTyping and Inheritance in C++**
- **A Mazing Problem**
- **Evaluation of Expressions**

Example of Selection Sort



Selection Sort Using Template

```
1.  template <class KeyType>
2.  void  sort (KeyType *a, int n)
3.  // sort the n KeyType a[0] to a[n-1] into nondecreasing order
4.  {
5.     for(int k=0; k < n; k++){
6.          int smallest = k;
7.          // find the smallest KeyType in a[k] to a[n-1]
8.          for (int j = k+1; j < n; j++){
9.               if(a[j] < a[smallest]) smallest = j;
10.         }
11.         KeyType temp = a[k]; a[k] = a[smallest]; a[smallest] = temp;
12.    }
13. }
14. main(){
15.    float  real_array[100];
16.    int    int_array[250];
17.    … // assume that the arrays have been initialized
18.    sort(real_array, 100);
19.    sort(int_array, 250);
20.    …
21. }
```

Concept of Template

- **A Template**
  - **also called parameterized types**
  - **It may be viewed as a variable which can be instantiated to any data type when a function or class is used**
- **Copy Constructor**
  - **is a special constructor which is invoked when an object is initialized with another object**
  - **e.g., *KeyType* temp=*a*[k];**
  - **might be overloaded if the parameterized type is not a pre-defined data type in C++**

Container Class

```
1.  class Bag
2.  {
3.  public:
4.      Bag (int MaxSize = DefaultSize); // constructor
5.      ~Bag();                  // destructor
6.      void Add(int);          // insert an integer into Bag
7.      void Delete(int &); // delete an integer from Bag
8.      Boolean IsFull();       // return TRUE if the bag is full; FALSE otherwise
9.      Boolean IsEmpty(); // return TRUE if the Bag is empty; FALSE otherwise
10. private:
11.     void Full();       // action when bag is full
12.     void Empty(); // action when the bag is empty
13.
14.     int *array;
15.     int  MaxSize; // size of array
16.     int  top;         // highest position in array that contains an element
17. };
```

Implementation of Bag

```
1.  Bag::Bag(int MaxBagSize): MaxSize (MaxBagSize) {
2.     array = new int[MaxSize]; top = -1;
3.  }
4.  Bag::~Bag() { delete [] array; }
5.  inline Boolean Bag::IsFull(){
6.        if(top == MaxSize – 1) return TRUE; else return FALSE;
7.  }
8.  inline Boolean Bag::IsEmpty(){
9.        if(top == -1) return TRUE; else return FALSE;
10. }
11. inline void Bag::Full(){ cout << "Bag is full" << endl; }
12. inline void Bag::Empty(){ cout << "Bag is empty" << endl; }
13. void  Bag::Add(int x) {  if(IsFull()) Full; else array[++top] = x;  }
14. int  *Bag::Delete(int &x){
15.       if( IsEmpty()) { Empty(); return(0); }
16.       int deletePos = top/2; x = array[deletePos];
17.       for(int k=deletePos; k<top; k++){
18.             array[k] = array[k+1]; // compact upper half of array
19.       top--; return(&x);
20. }
```

Parameterized Container Class

```
1.  template <class Type>
2.  class Bag
3.  {
4.  public:
5.     Bag (int MaxSize = DefaultSize); // constructor
6.     ~Bag();                  // destructor
7.     void Add(const Type&);    // insert an element into Bag
8.     Type *Delete(Type&);      // delete middle element from Bag
9.     Boolean IsFull();      // return TRUE if the bag is full; FALSE otherwise
10.    Boolean IsEmpty(); // return TRUE if the Bag is empty; FALSE otherwise
11. private:
12.    void Full();      // action when bag is full
13.    void Empty(); // action when the bag is empty
14.
15.    Type *array;
16.    int  MaxSize; // size of array
17.    int  top;        // highest position in array that contains an element
18. };
19. main(){
20.   Bag<int> a;    Bag<Rectangle> r;   …
21. }
```

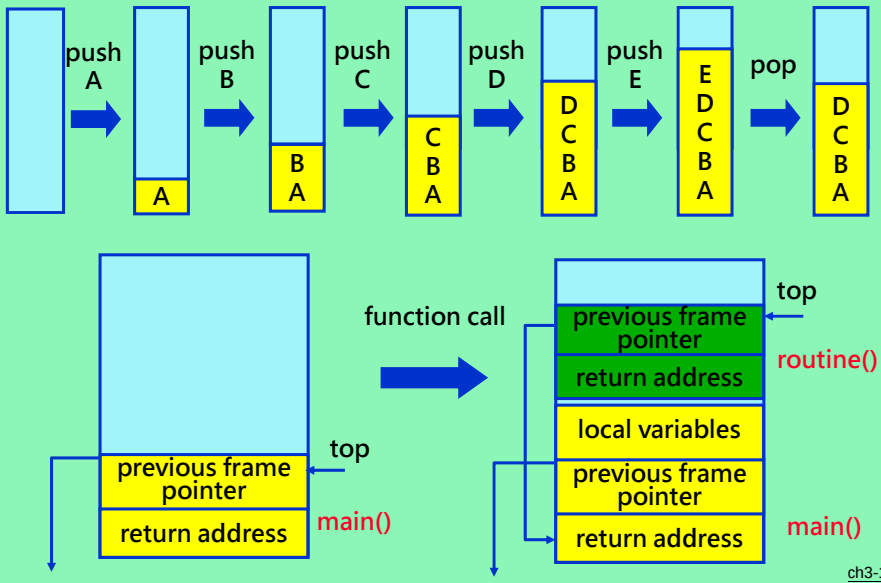
Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

Stack

- **Ordered List**
  - Suppose A = $a_0, a_1, \ldots, a_{n-1}$, where $n \geqq 0$
  - a_i is called an **atom**, or an **element**
- **Stack: Last-In First-Out (LIFO)**
  - is a special case of ordered list
  - the **additions** and **deletions** are made at one end called the *top*
  - E.g., Given a stack S = $(a_0, a_1, \ldots, a_{n-1})$
  - a_0 is the bottom element
  - a_{n-1} is the top element

Example: Stack Operation



ADT of Stack

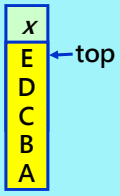
```
1.  template <class KeyType>
2.  class Stack
3.  {
4.  // objects: A finite ordered list of zero or more elements
5.  public:
6.     Stack (int MaxStackSize = DefaultSize);
7.     Boolean IsFull();      // return TRUE if Stack is full; FALSE otherwise
8.     Boolean IsEmpty(); // return TRUE if Stack is empty; FALSE otherwise
9.     void Add(const KeyType&);
10.            // if IsFull(), return 0;
11.            // else insert an element to the top of the Stack
12.    KeyType *Delete(KeyType&);
13.            // if IsEmpty(), then return 0;
14.            // else remove and return a pointer to the top element
15. private:
16.    int  top;
17.    KeyType *stack;
18.    int  MaxSize;
19. };
```

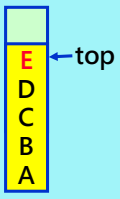
Member Functions of Stack

```
1.  template <class KeyType>
2.  Stack<KeyType>::Stack (int MaxStackSize) : MaxSize (MaxStackSize)
3.  {
4.          stack = new KeyType[MaxSize];
5.          top = -1;
6.  }
7.
8.  template <class KeyType>
9.  inline Boolean Stack<KeyType>::IsFull()
10. {  if(top == MaxSize – 1) return TRUE;   else return FALSE;  }
11.
12. template <class KeyType>
13. inline Boolean Stack<KeyType>::IsEmpty()
14. {    if (top == -1) return TRUE; else return FALSE;    }
15.
```


Push & Pop of Stack

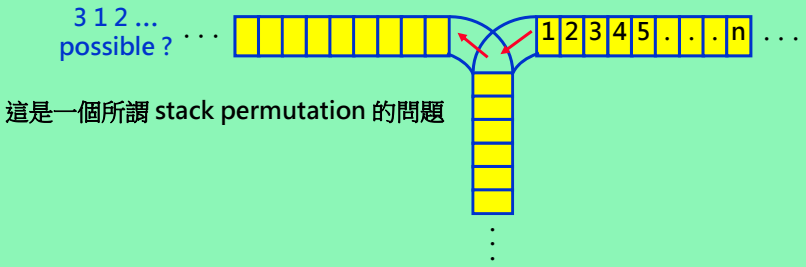
```
1.  template <class KeyType>
2.  void  Stack<KeyType>::Add(const KeyType& x)  // the push operation
3.  {
4.          if( IsFull() ) StackFull();
5.          else {
6.                  stack[++top] = x;
7.          }
8.  }
9.
10. template <class KeyType>
11. KeyType* Stack<KeyType>::Delete(KeyType& x) // the pop operation
12. {
13.         if (IsEmpty() ) StackEmpty();
14.         else {
15.                 x = stack[top--];   // assigned the item being deleted to x
16.                 return( &x );
17.         }
18. }
```





Railroad Switching System

- **Switching Rule**
  - **Initial: train 1, 2, …, *n* in the top right track segment**
  - **Movement:**
    - **(1) from top-right to the vertical segment one at a time**
    - **(2) from the vertical to the top-left segment one at a time**
    - **(3) The vertical segment operates like a stack**
- **Question: What output permutations are not possible?**



這是一個所謂 stack permutation 的問題

Outline

- **Templates in C++**
- **The Stack Abstract Data Type**
- **The Queue Abstract Data Type**
- **SubTyping and Inheritance in C++**
- **A Mazing Problem**
- **Evaluation of Expressions**

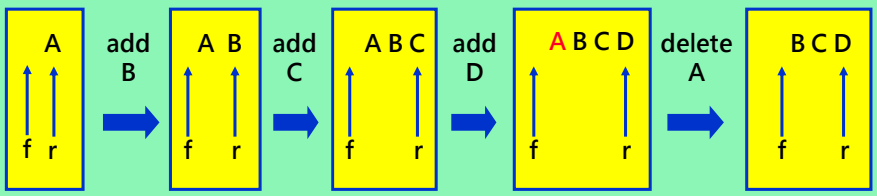
Queue

- **A Queue**
  - is an ordered list in which all **insertions** take place at one end and all **deletions** take place at the opposite end
  - is also called **First-In First-Out (FIFO)**
- **Example**
  - Given a queue Q = (a_0, a_1,…, a_{n-1})
  - a_0 is the **front** element, a_{n-1} is the **rear** element
  - a_i is **behind** a_{i-1} for $1 \leqq i \leqq n$

Data and Operations of Queue

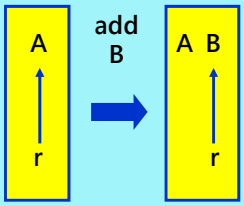
```
1.  template <class KeyType>
2.  class  Queue {
3.  public: …
4.  private:
5.     int  front;          // front: index of the first element to be retrieved
6.     int  rear;           // rear: index of the last element
7.     KeyType  *queue;     // KeyType array
8.     int  MaxSize;
9.  };
```

f = r = -1 initially



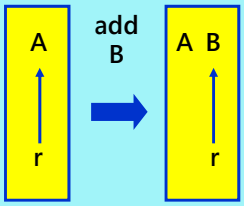
Member Functions of Queue

```
1.  template <class KeyType>
2.  Queue<KeyType>::Queue (int MaxQueueSize) : MaxSize (MaxQueueSize)
3.  {
4.      queue = new KeyType[MaxSize];
5.      front = rear = -1;
6.  }
7.
8.  template <class KeyType>
9.  inline Boolean Queue<KeyType>::IsFull()
10. {
11.    if( rear == MaxSize –1 ) return TRUE;
12.    else  return FALSE;
13. }
14.
15. template <class KeyType>
16. inline Boolean Queue<KeyType>::IsEmpty()
17. {
18.    if( front == rear ) return TRUE;
19.    else  return FALSE;
20. }
```



Add and Delete for Queue

```
1.  template <class KeyType>
2.  void Queue<KeyType>::Add (const KeyType& x)
3.  // add x to the queue
4.  {
5.      if( IsFull() ) QueueFull();
6.      else  queue[++rear] = x;
7.  }
8.
9.  template <class KeyType>
10. KeyType *Queue<KeyType>::Delete(KeyType& x)
11. // remove front element from the queue
12. {
13.    if( IsEmpty() ) { QueueEmpty(); return(); }
14.    x = queue[++front];
15.    return x;
16. }
```



Example: Job Scheduling

- **In Operating System**
  - **jobs are processed in the order they enter the system if no priority is set on jobs**

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	comments
-1	-1							queue is empty
-1	0	J1						Job 1 joins Q
-1	1	J1	J2					Job 2 joins Q
-1	2	J1	J2	J3				Job 3 joins Q
0	2		J2	J3				Job 1 leaves Q
0	3		J2	J3	J4			Job 4 joins Q
1	3			J3	J4			Job 2 leaves Q

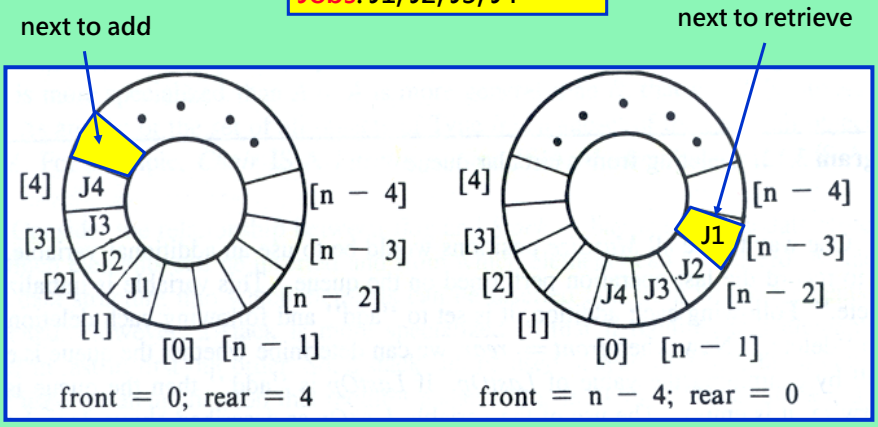
Worst-Case Scenario

front	rear	Q[0]	Q[1]	Q[2]	…	Q[n-1]	Next Operation
-1	n-1	J1	J2	J3	…	J_n	initial state
0	n-1		J2	J3	…	J_n	delete J1
-1	n-1	J2	J3	J4	…	J_{n+1}	add J_{n+1} (J2 to Jn are moved)
0	n-1		J3	J4	…	J_{n+1}	delete J2
-1	n-1	J3	J4	J5	…	J_{n+2}	add J_{n+2}

In the above job scheduling,
it takes n-1 steps to add a new job

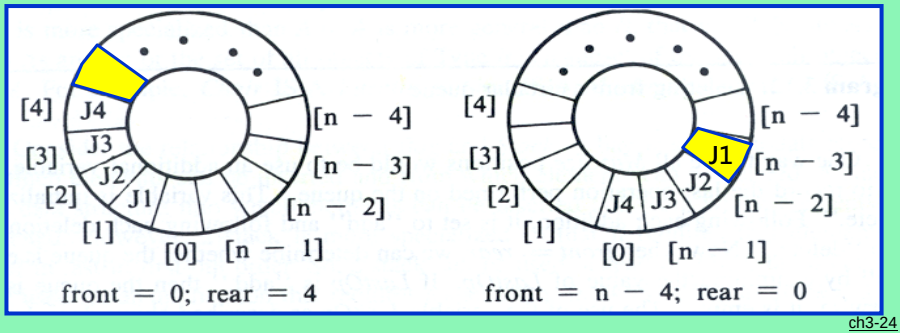
Circular Queue

Queue size: n
Jobs: J1, J2, J3, J4



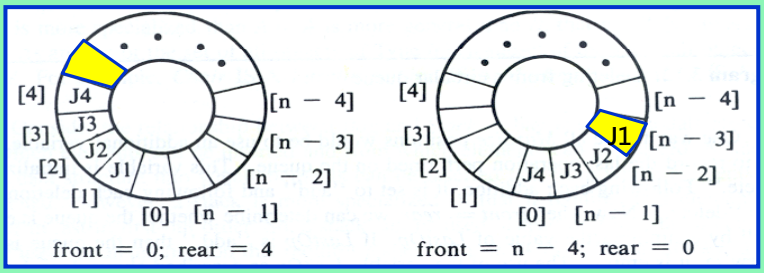
Add An Element to Circular Q

```
template <class KeyType>
void Queue<KeyType>::Add (const KeyType& x)
{
   int  new_rear=(rear+1)% MaxSize;
   if( front == new_rear)  QueueFull();
   else  queue[rear == new_rear] = x;
}
```



Delete An Element From Circular Q

```
template <class KeyType>
KeyType *Queue<KeyType>::Delete (KeyType& x)
// remove front element from queue
{
  if ( front == rear ) {  QueueEmpty(); return(0); }
  front = (front + 1) % MaxSize;
  x = queue[front];   return (&x);
}
```



DeQue

- **Definition**
  - A double-ended queue (Deque) is a linear list in which additions and deletions may be made at either end
- **Exercises**
  - Design a data representation that maps a deque into a one-dimensional array
  - Write algorithms to add and delete elements from either end of the queue

Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

SubType and Inheritance

- **IS-A Relation**
  - Chair IS-A furniture, Lion IS-A Mammal
  - Rectangle IS-A Polygon
  - Stack IS-A Bag
- **Inheritance**
  - Is used to express IS-A relationship between ADTs.
  - derived class IS-A base class
  - C++ provides a mechanism called public inheritance
    - Ex: class Stack: public Bag
  - The inherited members have the same level of access in the derived class as in the base class

Bag and Stack

```
class Bag
{
public:
    Bag (int MaxSize = DefaultSize); // constructor
    ~Bag(); // destructor
    virtual void Add(int);
    virtual int *Delete(int&);
    virtual Boolean IsFull();
    virtual Boolean IsEmpty();
protected:
    virtual void Full();
    virtual void Empty();
    int  *array;   int MaxSize;   int top;  // data members
};
```

(1) Interface of Bag will be reused in Stack
(2) The implementation of virtual functions may be re-defined in the derived class

```
class Stack : public Bag
{
public:
   Stack (int MaxSize = DefaultSize);
   ~Stack();
   int  *Delete(int&); // delete the element from stack
};
```

constructor, destructor cannot be reused

Example of Derived Class

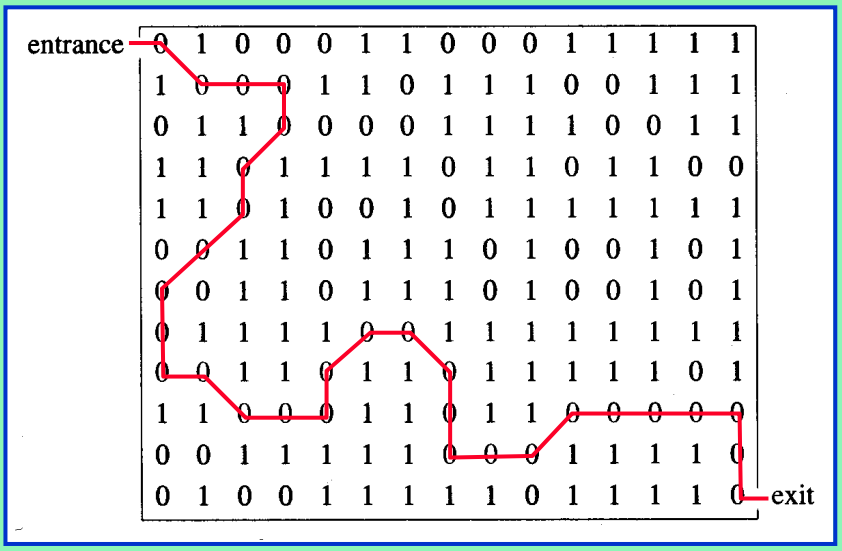
```
Stack::Stack(int MaxStackSize) : Bag(MaxStackSize) { }
// Constructor for Stack calls constructor for Bag

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

int *Stack::Delete(int& x)
{
    if ( IsEmpty() ) { Empty(); return(0); }
    x = array[ top-- ];
   return(&x);
}
```

Example: Stack s(3), int *x*
s.Add(1); s.Add(2); s.Add(3);
// Stack::Add is not defined, so use Bag::Add instead
s.Delete(*x*)
// uses Stack::Delete, which calls Bag::IsEmpty and Bag::Empty
because these have not been redefined in Stack

Outline

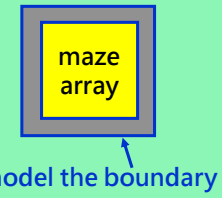
- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

An Example Maze



Data Structure

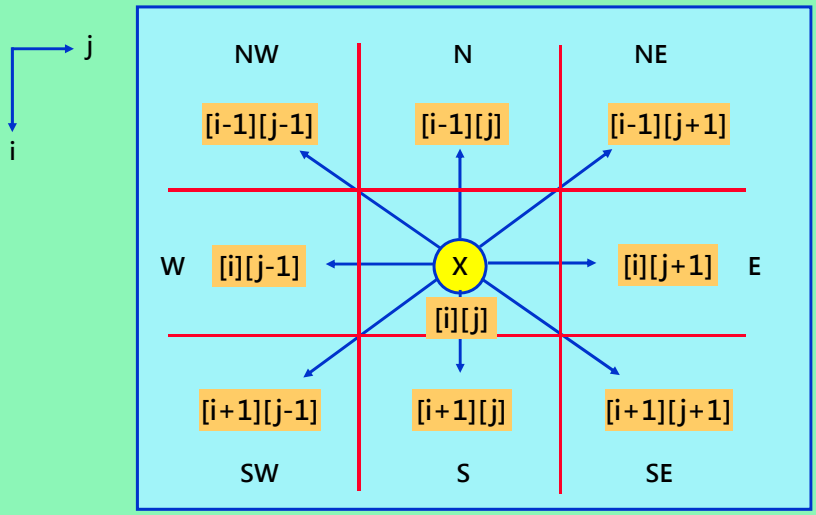
- **Maze**
  - Is represented as a two-dimensional array maze[i][j]
  - maze[i][j]=0: location that can be passed through
  - maze[i][j]=1: blocked location
  - Entrance: maze[0][0]
  - Exit: maze[m][p]
- **To model border condition**
  - The array is declared as maze[m+2][p+2]
  - I.e., the original maze array is surrounded by a border of ones



Strategy of Searching

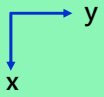
- **As a rat walks through the maze**
  - (1) He picks a valid move from the current position
    - E.g., starting from north and looking clockwise
  - (2) Put the selected move into a stack
    - So that he can return from a dead path
  - (3) He learns not to make the same mistake twice
    - Avoid getting into a cell visited before
    - A 2-dimensional array, mark[m+2][p+2] is used
    - The mark array records the cells visited before

Allowable Moves



Coordinates of the Next Move

The coordinates of the next move is computed by the following data structure




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

→ The SW of (i, j) will be (g, h) where
g = i + move[SW].x;
h = j + move[SW].y

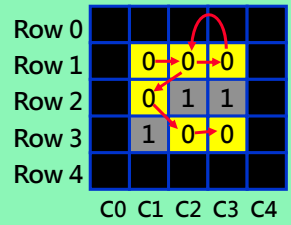
q	move[q].x	move[q].y
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

First Pass at Finding A Path Through a Maze

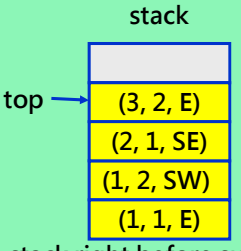
```
initialize stack to the maze entrance coordinates and direction east;
while (stack is not empty)
{
 (i,j,dir) = coordinates and direction deleted from top of stack ;
 while (there are more moves)
 {
  (g,h) = coordinates of next move ;
  if (( g == m ) && ( h == p )) success ;
  if ( (!maze [g ][h])  // legal move
            && (!mark [g ][h])  // haven't been here before
  {
        mark [g ][h] = 1 ;
        dir = next direction to try ;
        add (i,j,dir) to top of stack;
        i = g;  j = h;  dir = north;
  }
 }
}
cout << "no path found" << endl ;
```

An item of the stack:
struct items {
    int *x, y, dir*;
}

Example: A Mazing Problem



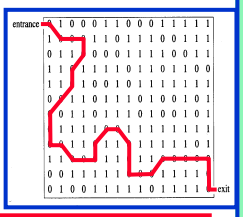
Current Position	Next Legal Move	Stack operation
(1, 1)	(1, 2, E)	Push (1, 1, E)
(1, 2)	(1, 3, E)	Push (1, 2, E)
(1, 3)	No legal move	Pop to backtrack
(1, 2)	(2, 1, SW)	Push (1, 2, SW)
(2, 1)	(3, 2, SE)	Push (2, 1, SE)
(3, 2)	(3, 3, E) success!	Pop out the entire stack



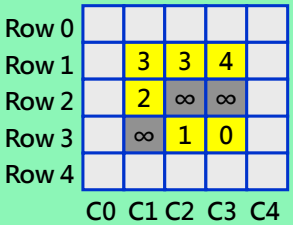
stack right before success

Complete path:
(3, 3) ← (3, 2, E) ← (2, 1, SE) ← (1, 2, SW) ← (1, 1, E)

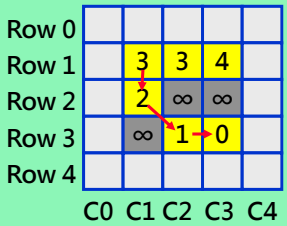
```
void path(int m, int p)
// Output a path (if any) in the maze; maze[0][i] = maze [m +1][i] =
// maze [j][0] = maze [j ][p +1] = 1, 0 ≤ i ≤ p +1, 0 ≤ j ≤ m+1.
{
   // start at (1,1)
   mark[1][1] = 1 ;
   Stack<items> stack(m*p) ;
   items temp ;
   temp.x = 1 ; temp.y = 1 ; temp.dir = E ;
   stack.Add(temp) ;
   while (!stack.IsEmpty()) // stack not empty
   {
      temp = *stack.Delete (temp) ; // unstack
      int i = temp.x ; int j = temp.y ; int d = temp.dir ;
      while (d < 8) // move forward
      {
         int g = i + move[d].a ; int h = j + move[d].b ;
         if ((g == m) && (h == p)) { // reached exit
            // output path
            cout << stack ;
            cout << i << " " << j << endl ; // last two squares on the path
            cout << m << " " << p << endl ;
            return;
         }
         if ((!maze[g][h]) && (!mark[g][h])) { // new position
            mark[g][h] = 1 ;
            temp.x = i ; temp.y = j ; temp.dir = d+1 ;
            stack.Add(temp) ; // stack it
            i = g ; j = h ; d = N ; // move to (g,h)
         }
         else d++ ; // try next direction
      }
   }
   cout << "no path in maze " << endl ;
}
```



Backward Search



after step1



after step2

Backward_search_algorithm
{
Step 1:
compute the **distance** to the destination of each node by **wave-front propagation**
Step 2:
find a **shortest path** from the source to the destination node by **picking up the nodes with a shortest distance**
}

Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

Types of Expression

- **Arithmetic Expression**
  - For example: X = A/B – (C + D * E – A * C)
  - The evaluation of this expression is critical in enabling high-level programming
  - An expression consists of
    (1) Operands: A, B, C, D, E
    (2) Operator: plus, minus, times, and divide
    (3) Delimiter: like parenthesis "(", ")"
- **Boolean Expression**
  - The result in TRUE or FALSE
  - Use relational and logical operators
    - <, <=, ==, >, >=, &&, ||, !

Priority of Operator

- **Priority**
  - **The order of operations to be carried out in an expression**
  - **Different order leads to different results**

```
X = A/B – C + D*E – A*C
  = ((4/2) –2) + (3*3) – (4*2)
  = 0 + 9 – 8
  = 1
```

```
X = A / B – C + D*E – A*C
  = ( 4 / (2 – 2 + 3) * (3 – 4) * 2 )
  = (4/3) * (-1) * 2
  = -2.6666
```

Times and divide
have higher priorities
→ Default

Plus and Minus
have higher priorities
→ Not default

Priority of Operations in C++

Evaluation of operators of the same priority will proceed from left to right
E.g., A/B*C → (A/B) * C

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

Postfix Notation

- **Compiler**
  - **Translates an expression into a sequence of machine codes**
  - **It first re-writes the expression into a form called postfix notation**
- **Infix notation**
  - **The operators come in-between operands**
- **Postfix notation**
  - **The operators appear after its operands**
- **Example**
  - **Infix: A / B – C + D * E – A * C**
  - **Postfix: A B / C – D E * + A C * –**

Evaluation of Postfix Notation

- **Scanning the notation from left to right**
- **Store temporary result in T_i, $i \geqq 1$**

Original expression: AB/C-DE*+AC*-	
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

The Virtues of Postfix Notation

- **Evaluation is easier on postfix notation**
  - **The need for parenthesis is eliminated**
  - **The priority of operations is no longer relevant**
  - **Evaluation of each operator**
    - **(1) Pop correct number of operands from the stack**
    - **(2) Perform the operation**
    - **(3) Push the results onto the stack**

Evaluation Algorithm

```
void evaluation(expression e)
// Evaluate the postfix expression e. It is assumed that the last
// token ( a token is either an operator, operand, or '#') in e is '#'
// A function NextToken is used to get the next token from e.
{
   Stack<token> stack; // initialize the stack
   for( token x = NextToken(e); x != '#'; x = NextToken(e) ){
        if( x is an operand )  stack.Add(x)  // add to stack
        else { // operator
              pop-up correct number of operands for operator x;
             perform the operation x and store the result onto
             the stack;
        }
   }
}
```

Conceptual Infix to Postfix

- **Algorithm**
  - **(1) Fully parenthesize the expression**
  - **(2) Move all operators so that they replace their corresponding right parentheses**
  - **(3) Delete all parentheses**

Example: A / B – C + D * E – A * C
→ ((((A / B) – C) + (D * E)) – (A * C))

→ A B / C – D E * + A C * –

Ex1: From Infix to Postfix

- **Translate A+B*C to ABC*+**

Next token	Stack	Output
None	Empty	None
A	Empty	A
+	+	A
B	+	AB
*	+ *	AB
The operator * has a higher priority than +, so it is placed on top of +		
C	+ *	ABC

(1) All operands go directly to the output
(2) When the input tokens are exhausted
→ All operators in the stack will be taken off

Ex2: From Infix to Postfix

- **Translate A*(B+C)/D to ABC+*D/**

Next token	Stack	Output
None	Empty	None
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
/	/	ABC+*
D	/	ABC+*D
'#' (ending character)	Empty	ABC+*D/

Priority-Based Stack Operation

- **Left Parenthesis**
  - Behaves as an operator with high priority when it is not in the stack → incoming priority (icp) = 0
  - Behaves as one with low priority when it is in the stack → in-stack priority (isp) = 8
  - Only the matching right parenthesis can get an in-stack left parenthesis unstacked
- **Summary**
  - Operators are taken out of the stack as long as their in-stack priority is numerically smaller than or equal to the in-coming priority of the new operator
  - Assuming that the icp('#') = 8 (lowest)

Algorithm of Infix to Postfix

```
void postfix (expression e)
// Output the postfix form of the infix expression e. NextToken
// and stack are as in function eval (Program 3.18). It is assumed that
// the last token in e is '#.' Also, '#' is used at the bottom of the stack
{
   Stack<token> stack; // initialize stack
   token y ;
   stack.Add('#');
   for (token x = NextToken(e) ; x != '#' ; x = NextToken(e))
   {
      if (x is an operand)  cout << x ;
      else if (x == ')')  // unstack until '('
         for (y = *stack.Delete (y); y != '('; y = *stack.Delete (y)) cout << y ;
      else { // x is an operator
         for (y = *stack.Delete (y); isp (y) <= icp (x); y = *stack.Delete (y)) cout << y ;
         stack.Add(y);   // restack the last y that was unstacked
         stack.Add(x);
      }
   }

   // end of expression; empty stack
   while (!stack.IsEmpty()) cout << *stack.Delete(y) ;
} // end of postfix
```

higher value means lower priority

Using Stack in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL Stack 網頁: http://www.cppreference.com/wiki/stl/stack/start

Example: the following code uses empty() as the stopping condition on a while loop to clear a stack and display its contents in reverse order:

```
stack<int> s;
for( int i = 0; i < 5; i++ )  { s.push(i); }
while( !s.empty() )  { cout << s.top() << endl; s.pop(); }
```

The End of Chapter 3

Next Topic:

Linked Lists