Stacks and Queues

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

Topics

- **Templates in C++**
- Stack Abstract Data Type
- Queue Abstract Data Type
- Subtyping and inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

Templates in C++

- Make classes and functions more reusable
- Without using templates
  - Selection sort

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

Selection Sort

n=8 i=0
j = 0 k=1

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

sorted

Selection Sort

n=8 i=0
j = 1 k=2

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

sorted

Selection Sort

n=8 i=0
j = 1 k=3

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

sorted

Selection Sort

n=8 i=0
j = 1 k=4

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5
sorted								

Selection Sort

n=8 i=0
j = 4 k=5

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5
sorted								

Selection Sort

n=8 i=0
j = 4 k=6

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

sorted

Selection Sort

n=8 i=0
j = 4 k=7

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

sorted

Selection Sort

n=8 i=0
j = 4 k=8

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted

Selection Sort

n=8 i=1
j =1 k=2

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted

Selection Sort

n=8 i=1
j =1 k=3

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted

Selection Sort

n=8 i=1
j =1 k=4

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted

Selection Sort

n=8 i=1
j =1 k=5

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted

Selection Sort

n=8 i=1
j =1 k=8

index	0	1	2	3	4	5	6	7
value	1	2	8	6	3	7	4	5

sorted (indices 0–1)

Selection Sort

n=8 i=2
j =2 k=8

index	0	1	2	3	4	5	6	7
value	1	2	3	6	8	7	4	5

sorted

Templates in C++ (Cont.)

- To sort an array of floating point numbers
  - In line 1, int *a → float *a
  - In line 2, integers → floats
  - In line 11, int → float
- This process repeats for arrays of other data types
- Template (parameterized type)
  - variable that can be instantiated to any data type

Templates Functions

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

❖ Program 3.1 : Selection sort using templates

Templates Functions (Cont.)

```
float farray[100];
int intarray[250];
...
// assume that the arrays are initialized at this point
sort(farray, 100);
sort(intarray, 250);
```

❖ Program 3.2 : Code fragment illustrating template instantiation

Copy Constructor

- Invoked when an object is initialized with another object
- A type specifier specifies the type of the object in an initialization

Operators in Functions using Templates

- <, =, copy constructor
- Automatically defined for int and float
- Not pre-defined for user-defined data types
- May be overloaded in a manner that is consistent with their usages
- If = and copy constructor for a user-defined class are not defined by the user, the compiler creates default implementations

Template Function Example

- Changing the size of a 1-dimensional 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];
    int number = min(oldSize, newSize);
    copy(a, a+number, temp);
    delete [] a;
    a = temp;
}
```

❖ Program 3.3 : Template function to change the size of a 1−dimensional array

Using Templates to Represent Container Class

- Container class
  - A data structure that stores a number of data objects
  - Objects can be added or deleted
- Bag
  - Can have multiple occurrences of the same object
  - The position of an element is immaterial
  - Any element can be removed for a delete operation

Bag of integers

- C++ array implementation of Bag of integers
  - insertion : to the first available position
  - deletion : the element in the middle position and then compact upper half of array

Bag of integers (Cont.)

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

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

  void Push(const int);
  void Pop();

private:
   int *array;
   int capacity;                   // size of array
   int top;                        // highest position in array that contains an element
```

❖ Program 3.4 : Definition of class Bag containing integers

Bag of integers (Cont.)

```
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] = x;
}
```

Bag of integers (Cont.)

```
void Bag::Pop() {
    if(Is Empty()) throw "Bag is empty, cannot delete";
    int deletePos = top / 2;
    copy(array + deletePos + 1, array + top + 1, array + deletePos);
              // compact array
    top--;
}
```

❖ Program 3.5 : Implementation of operations on Bag

Template class definition for Bag

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

  int Size() const;
  bool IsEmpty() const;
  T& Element() const;

  void Push(const int);
  void Pop();

private:
   T *array;
   int capacity;
   int top;
};
```

❖ Program 3.6 : Definition of template class Bag

Template Class Instantiation

- Instantiation of template class Bag
  - Bag<int> a;
  - Bag<Rectangle> r;

Implementation of Operations of Template Class 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; }

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

Implementation of Operations of Template Class Bag (Cont.)

```
template <class T>
void Bag<T>::Pop() {
    if(Is Empty()) throw "Bag is empty, cannot delete";
    int deletePos = top / 2;
    copy(array + deletePos + 1, array + top + 1, array + deletePos);
            // compact array
    array[top--].~T();
}
```

❖Program 3.7 : Implementation of operations on Bag

Stack Abstract Data Type

- Stack
  - An ordered list in which insertions and deletions are made at one end called the top
  - $S=(a_0, \ldots, a_{n-1})$
    - a_0 : bottom element
    - a_{n-1} : top element
    - a_i is on top of a_{i-1}, $0<i<n$
- Last-In-First-Out(LIFO) list

Stack Abstract Data Type (Cont.)

- Add A, B, C, D, E, then delete E.

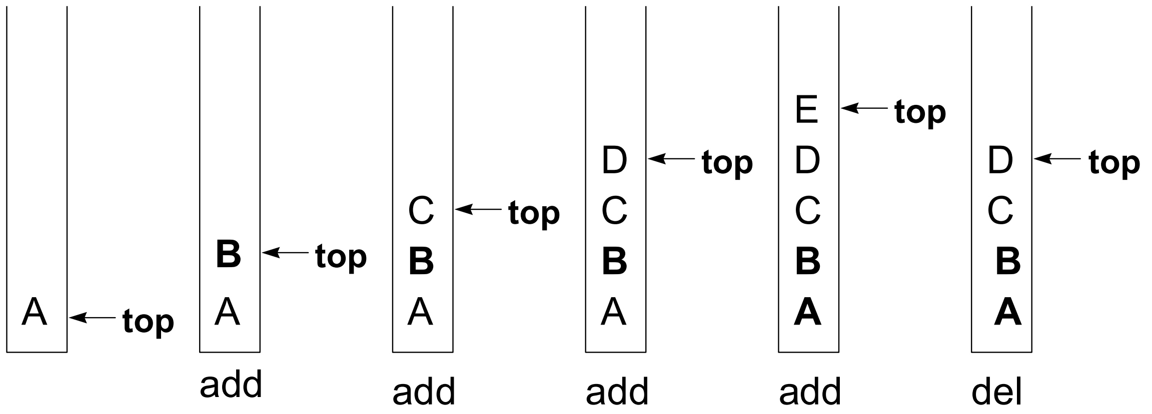


Figure 3.1 : Inserting and deleting elements in a stack

System Stack

- Used by a program at run time to process function calls
- When a function is invoked, the program creates a stack frame, and places it on top of the system stack
  - previous frame pointer
  - return address
  - local variables
  - parameters
- When a function terminates, its stack frame is removed
- A recursive call is processed in the same way

System Stack (Cont.)

- A main function invokes function a1
  - (a) : Before a1 is invoked
  - (b) : After a1 has been invoked
  - fp : current stack frame pointer

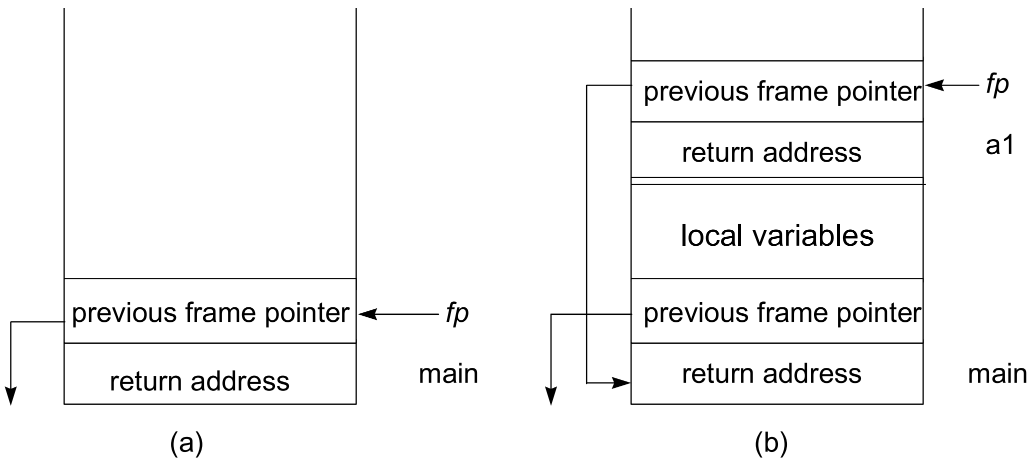


Figure 3.2 : System stack after function call

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 maximum size is stackCapacity

   Boolean IsEmpty() const;
   // if number 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 ADT

- Use an one-dim array stack[stackCapacity]
- Bottom element is stored in stack[0]
- Top points to the top element
  - initially, top=-1 for an empty stack
- Data member declarations

```
Private:
   int top;
   T *stack;
   int capacity;
```

Implementation of Stack ADT (Cont.)

- constructor definition

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

- member function IsEmpty()

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

Implementation of Stack ADT (Cont.)

- member function Top()

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

- 'StackFull' and 'StackEmpty' functions depend on the particular application

Implementation of Stack ADT (Cont.)

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

❖Program 3.8 : Adding to a stack

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

❖Program 3.9 : Deleting from a stack

Queue Abstract Data Type

- Queue
  - An ordered list in which all insertions take place at one end and all deletions take place at the opposite end
  - $Q=(a_0, a_1, ..., a_{n-1})$
    - a_0 : front element
    - a_{n-1} : rear element
    - a_i is behind a_{i-1}, $0<i<n$
  - First-In-First-Out(FIFO) list

Queue Abstract Data Type

- Inserting A, B, C, D, E, then delete an element

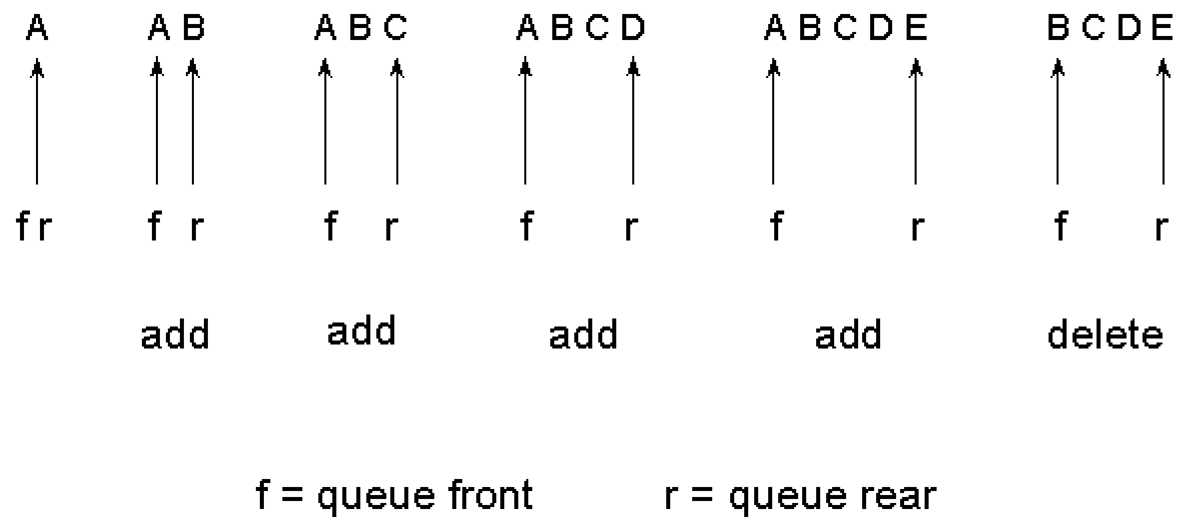


Figure 3.4 : Inserting and deleting elements in a queue

Queue ADT

```
template <class T>
class Queue
{ // A finite ordered list with zero or more elements.
public:
   Queue (int queueCapacity = 10);
   // Create an empty queue whose maximum size is queueCapacity

   Boolean IsEmpty() const;
   // if number 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 ADT

- Use an one-dim array
- two variables
  - front : one less than the position of the first element
  - rear : the position of the last element
- data member declaration

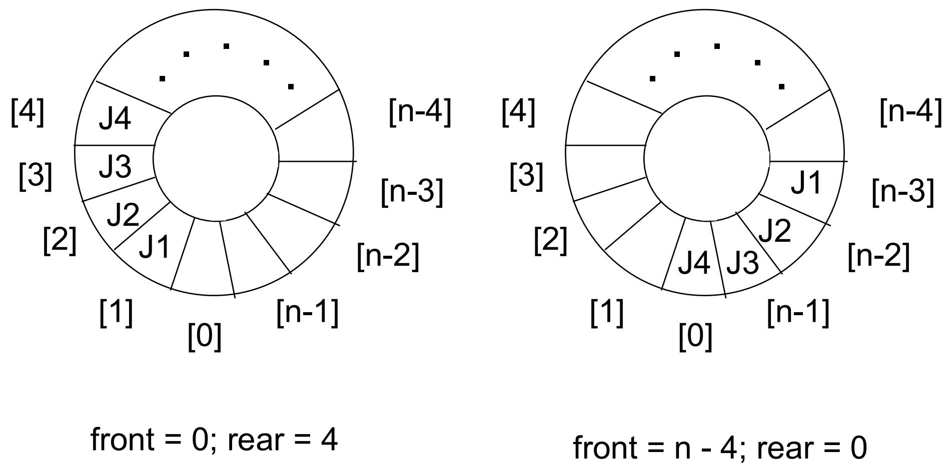
```
Private:
  T* queue;
  int front,
      rear,
      capacity;
```

Circular Queue

- the array queue[MaxSize] as circular
  - When rear==MaxSize-1, the next element is entered at q[0] if it is empty
- front : point one position counterclockwise from the first element
- a maximum of MaxSize-1 rather than MaxSize
  - to determine whether the queue is full or empty when front==rear
- Initially front==rear==0

Circular Queue (Cont.)

- Circular queue of MaxSize=n elements and four- jobs : J1, J2, J3, J4



Implementation of Circular Queue

- Constructor Definition

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

- Member function IsEmpty()

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

Implementation of Circular Queue (Cont')

- Member function Front()

```
template <class T>
inline T& Queue<T>::Front()
{
    if(IsEmpty()) throw "Queue is empty. No front element";
    return queue[(front+1) % capacity];
}
```

- Member function Rear()

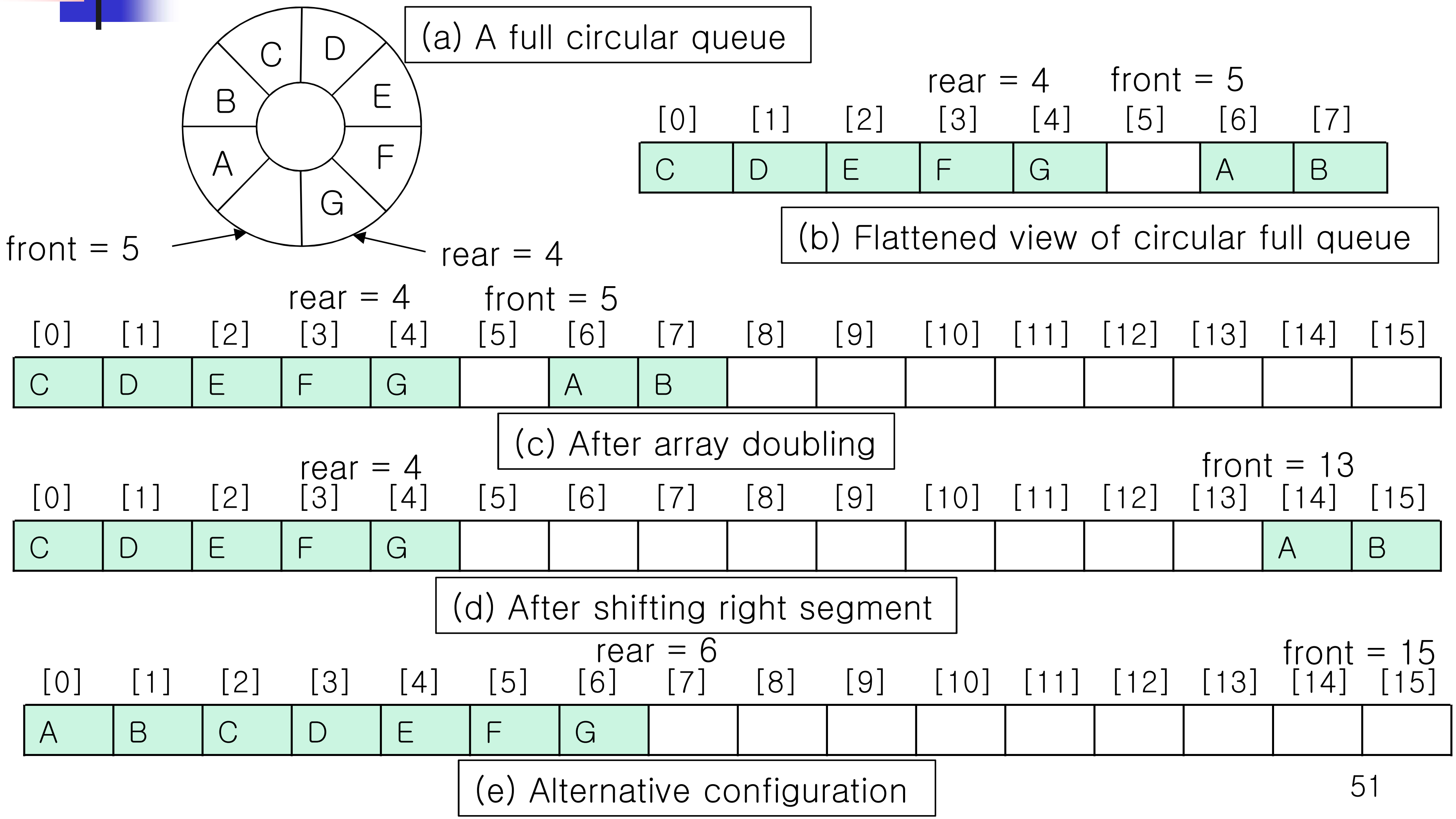
```
template <class T>
inline T& Queue<T>::Rear()
{
    if(IsEmpty()) throw "Queue is empty. No rear element";
    return queue[rear];
}
```

Implementation of Circular Queue (Cont.)

- Member function Push()

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

Doubling Queue Capacity



Doubling Queue Capacity

```
// allocate an array with twice the capacity
T* newQueue = new T[2*capacity];

// copy from queue to newQueue
int start = (front + 1) % capacity;
if(start < 2)
     // no wrap around
     copy(queue + start, queue + start + capacity – 1, newQueue);
else
{ // queue wraps around
     copy(queue + start, queue + capacity, newQueue);
     copy(queue, queue + rear + 1, newQueue + capacity – start);
}

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

❖Program 3.11 : Doubling queue capacity

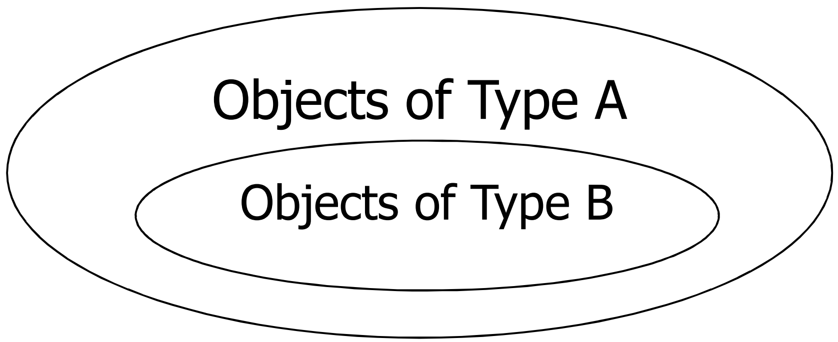
Implementation of Circular Queue (Cont.)

- Member function Pop()

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

Subtyping and Inheritance in C++

- Inheritance
  - subtype relationships between ADTs
  - IS-A relationship
  - B IS-A A
    - B is more specialized than A
    - A is more general than B
  - examples
    - Chair IS-A Furniture
    - Lion IS-A Mammal
    - Stack IS-A Bag



Subtyping and Inheritance in C++ (Cont.)

- C++ mechanism
  - public inheritance
  - base class : more general ADT
  - derived class : more specialized ADT
  - The derived class inherits all the non-private (protected or public) members (data and functions) of the base class
  - Inherited members have the same level of access in the derived class as they did in the base class
  - reuse
    - interface : Inherited member functions have the same prototypes
    - implementation : Only functions to override the base class implementation are reimplemented

Bag and Stack

```
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;
};
```

Bag and Stack (Cont.)

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

❖Program 3.13 : Definition of Bag and Stack

Stack Operations

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

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

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

void Stack::Pop()
{
   if(IsEmpty()) throw "Stack is empty. Cannot delete.";
   top--;
}
```

❖Program 3.14 : Implementation of Stack operations

Example

```
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);
//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::IsEmpty because IsEmpty has not been
// redefined in Stack.

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

s.Element; // uses Bag::Element
```

Example (Cont.)

- Result : b=<1, 3>, s=<1, 2>
- Queue
  - Subtype of Bag
    - Elements are deleted in FIFO order

A Mazing Problem

entrance	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	
	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	
	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	
	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	
	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	
	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1	
	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	
	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	
	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	exit

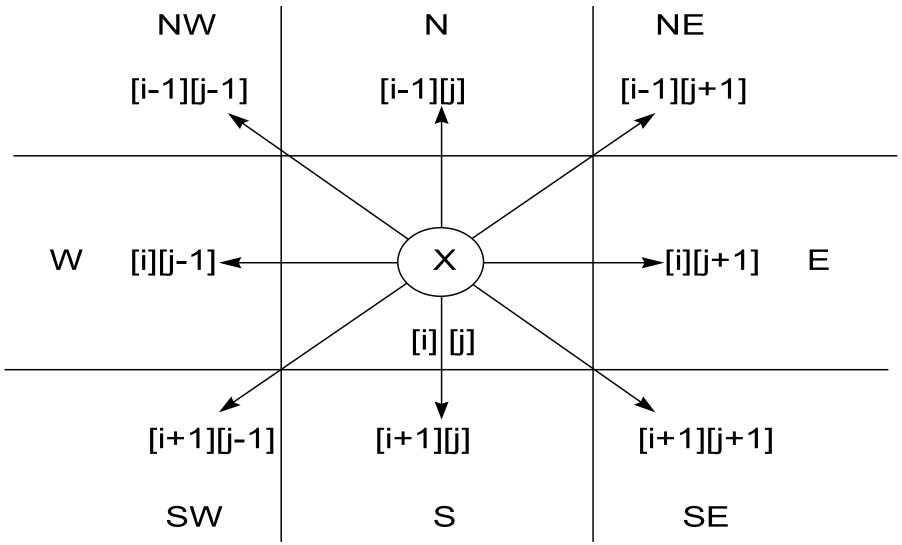
An example maze

A Mazing Problem

- Maze
  - Represent by a two-dim array, maze[i][j], where $1 \leq i \leq m$ and $1 \leq j \leq p$
  - 1 means a blocked path
  - 0 means a path
  - Starts at maze[1][1]
  - Exits at maze[m][p]

Possible moves

- Possible moves from X=maze[i][j]



- To avoid checking for border conditions
  - Surround the maze by a border of ones
  - Declare the array as maze[m+2][p+2]

Possible moves (Cont.)

- Table to predefine the possible directions to move
  - Necessary data types

```
struct offsets
{
   int a, b;
};


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

Possible moves (Cont.)

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

Figure 3.13 : Table of moves

Possible Moves (Cont.)

- To find maze[g][h] that is SW of maze[i][j]

  g=i+move[SW].a

  h=j+move[SW].b

- mark[m+2][p+2]
  - To prevent from going down the same path twice

Possible Moves (Cont.)

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

❖Program 3.15 : First pass at finding a path through a maze

Maze Algorithm

- Arrays maze, mark, move are global
- Stack is a stack of items

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

```
void path(const int m, const 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(1, 1, E);
          // set temp.x, temp.y, and temp.dir
   stack.Push(temp);

   while (!stack.IsEmpty())
   {// stack not empty
      temp = stack.Top();
      stack.Pop();
      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;
}
```

Overloading Operator <<

```
template <class T>
ostream& operator<<(ostream& os, Stack<T>& s)
{
    os << "top = " << s.top << endl;
    for(int i=0; i<=s.top; i++)
        os << i << ":" << s.stack[i] << endl;
    return os;
}

ostream& operator<<(ostream& os, items& item)
{
    return os << item.x << "," << item.y << "," << item.dir;
}
```

❖Program 3.17 : Overloading operator<<

Analysis of path Function

- Paths are never taken twice
- Each iteration of the inner while loop takes constant time
- If the number of zeros in maze is z, at most z positions can get marked
- Since z is bounded by mp, the time complexity is O(mp)

Evaluation of Expressions

- Expressions
  - Operators
    - arithmetic operators
      - basic : +, -, *, /
      - other : unary minus, %
    - relational operators :
      - <, <=, ==, <>, >=, >
    - logical operators : &&, ||, !

Evaluation of Expressions (Cont.)

- Precedence

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

Figure 3.15 : Priority of operators in C++

Postfix Notation

- Notations
  - Infix notation
    - A*B/C
  - Postfix notation
    - AB*C/
  - Prefix notation
    - /*ABC
- Example
  - Infix : A/B-C+D*E-A*C
    =(((A/B-C)+(D*E))-(A*C)
  - Postfix : AB/C-DE*+AC*-

Postfix Notation (Cont.)

operations	postfix
$T_1 = A / B$	T_1 C − D E * + A C * −
$T_2 = T_1 - C$	T_2 D E * + A C * −
$T_3 = D * E$	T_2 T_3 + A C * −
$T_4 = T_2 + T_3$	T_4 A C * −
$T_5 = A * C$	T_4 T_5 −
$T_6 = T_4 - T_5$	T_6

Figure 3.16 : Postfix evaluation

Postfix Notation (Cont.)

- Virtues of postfix notation
  - The need for parentheses is eliminated
  - The priority of the operators is not relevant
    - The expression is evaluated by making a left to right scan

```
void eval(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. The function uses the stack
{
   Stack<token> stack; // initialize stack
   for(Token x = NextToken(e); x != '#'; x = NextToken(e))
      if (x is an operand) stack.Push(x) // add to stack
      else { // operator
      remove the correct number of operands for operator x from stack; perfor
m the operation x and store the result (if any) onto the stack;
      }
} // end of eval
```

❖Program 3.18 : Evaluating postfix expressions

Infix to Postfix

- Steps
  1. Fully parenthesize the expression
  2. Move all operators so that they replace their corresponding right parentheses
  3. Delete all parentheses
- A / B - C + D * E - A * C
  - step1 yields
    - ((((A/B)-C)+(D*E))-(A*C))
  - step 2 and 3 yield
    - AB/C-DE*+AC*-

Infix to Postfix (Cont.)

- To handle the operators, we store them in a stack
- E.g., A+B*C to yield ABC*+

next token	stack	output
none	empty	none
A	empty	A
+	+	A
B	+	AB

- At this point the algorithm must determine if * gets placed on top of the stack or if the + gets taken off.
  - Since * has higher priority, we should stack *, producing

*	+*	AB
C	+*	ABC

Infix to Postfix (Cont.)

- Now the input expression is exhausted, output all remaining operators in the stack
  - ABC*+

Infix to Postfix (Cont.)

- e.g.,. A*(B+C)*D yield ABC+*D*

next token	stack	output
none	empty	none
A	empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	ABC
C	*(+	ABC

- At this point, we want to unstack to the corresponding left parenthesis and then delete the left and right parentheses

)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
done	empty	ABC+*D*

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

❖Program 3.19 : Converting from infix to postfix form

Analysis of Postfix

- It makes only a left-to-right pass across the input
- The time spent on each operand is O(1)
- Each operator is stacked and unstacked at most once
- The time spent on each operator is also O(1)
- Thus, total time is $\theta(n)$

Job Scheduling

- Sequential queue
  - The sequential representation of a queue has pitfalls

job queue by an operating system

front	rear	Q[0]	[1]	[2]	[3]	[4]	[5]	[6]	...	Commnets
−1	−1		queue		empty					initial
−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
0	3			J3	J4					job 2 leaves Q

Insertion and deletion from a queue

Job Scheduling (Cont.)

- As jobs enter and leave the system, the queue gradually shifts to the right
  - eventually rear=MaxSize-1, the queue is full
- QueueFull() should move the entire queue to the left
  - first element at queue[0]
  - front=-1
  - recalculate rear
  - Worst-time complexity is O(MaxSize)
- Worst case alternate requests to delete and add elements

Job Scheduling (Cont.)

- Queue example

front	rear	q[0]	[1]	[2]	…	[n−1]	Next Operation
−1	n−1	J1	J2	J3	…	Jn	initial state
0	n−1		J2	J3	…	Jn	delete J1
−1	n−1	J2	J3	J4	…	Jn + 1	add Jn + 1 (jobs J2 through Jn are moved)
0	n−1		J3	J4	…	Jn + 1	delete J2
−1	n−1	J3	J4	J5	…	Jn + 2	add Jn + 2