Abstract Data Type (ADT)

Abstraction is the process to reduce the information content of a concept, typically to retain only information which is relevant for a particular purpose.

In computer science, abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details.

In OO-programming, we try to separate the logical properties of an object from its internal implementation details.

For example, the standardized user interface of an Android phone is a logical property of the device.

The construction of the physical Android phone is the implementation details.

From the point of view of the user, you only need to know the logical property (i.e. the user interface) of the device when you are using the phone, you don't need to know its internal implementation details.

ADT is a tool that allows the programmer to separate the logical properties of a data structure from its implementation details.

An ADT has 3 components:

- name of the ADT, called the type name
- the set of values belonging to the ADT, called the domain
- the set of operations on the data (corresponds to the member functions in the implementation)

Abstract Data Type : Stack

A stack is a list of homogeneous elements in which the addition and deletion of elements occur only at one end, called the top of the stack.

A stack is also called a Last In First Out (LIFO) data structure.

Operations on a stack:

- initialize: initialize the stack to an empty state
- size: determine the number of elements in the stack
- empty: determine if the stack is empty
- top: retrieve the value of the top element
- push: insert element at the top of stack
- pop: remove top element

In C++, we can define an ADT using an abstract class. In our discussion, I will try to follow the notations used in the C++ STL (Standard Template Library).

```
template<class Type>
class stackADT
{
public:
   virtual void initialize() = 0; //pure virtual function
   //remark: the initialize() function is not part of the
   //        C++ STL. The initialization can be taken care
   //        of by the constructor.

   virtual int size() const = 0;
   virtual bool empty() const = 0;

   virtual Type& top() const = 0; //return reference
                                  //of the top element

   virtual void push(const Type& item) = 0;//reference

   virtual void pop() = 0;
   //Note that in the C++ STL, the pop function does not
   //return the (old) top element that is removed.

   //Remark: in Java, the pop method of the class Stack
   //        will return the removed element.
}
```

Remarks:

In the textbook, the author includes a function `full()` in the `stackADT`.
The function `full()` returns `true` if the stack is full.

However, `full()` is not part of the `stack` class in C++ STL.
It is not required if the stack is implemented using linked list.

Basically in the high level applications using stack, usually we only need to check if the stack is empty.
If we call the `top()` function on an empty stack, an underflow exception occurs.

The `top()` function in the STL returns the top element by reference. It is different from the example given in the textbook, where the `top()` function returns the top element by value.

We shall first discuss some examples on the uses of stack in algorithm design, and then come back to discuss the internal implementation of the stack ADT.

One common use of stack is for the simulation of recursion, i.e. converting a recursive algorithm to an equivalent non-recursive algorithm using a stack.
We shall talk about this when we discuss the tree data structure.

In the following discussion, we assume the concrete class stack is defined:

```
template<class Type>
class stack: public stackADT<Type>
{
public:
   void initialize()
   {
      //detailed implementation
   }

   int size()
   {
      //detailed implementation
   }

   Type& top()
   {
      //detailed implementation
   }

   ...

   //other member functions, constructors and destructor

}
```

Example codes:

```
#include <stack> //class stack in the C++ STL

int gcd(int m, int n)
{
   int r;

   while (r = m % n)
   {
      m = n;
      n = r;
   }
   return n;
}


int main()
{
   stack<int> s;

   int i = 10;
   s.push(i);  //push i into stack

   s.push(20); //push 20 into stack

   int j = s.top(); //j = 20

   s.top() -= 5; //value of top element is changed to 15


   s.push(gcd(i, j)); //push the return value of gcd()

}
```

Infix and postfix expression formats

- In infix format, the binary operator is placed in between the 2 operands.
- The order of evaluation is determined by the precedence relation of the operators and parentheses, if any.
- The order of precedence is
  exponentiation
  multiplication, division
  addition, subtraction
- In postfix format, the operator is placed after the 2 operands. The order of evaluation is the same as the order in which the operators appear in the postfix expression.
- In the examples shown below, $ represents the exponentiation operator.

Infix	Postfix
A + B	AB +
A + B − C	AB + C −
(A + B) * (C − D)	AB + CD − *
A $ B * C − D + E / F / (G + H)	AB $ C * D − EF / GH + / +
((A + B) * C − (D − E)) $ (F + G)	AB + C * DE − − FG + $
A − B/(C * D $ E)	ABCDE $ * / −

- The advantage of postfix format is that it requires no parentheses.

Algorithm to evaluate a postfix expression (pseudo code)

To simplify the discussion, we only consider the binary operators, e.g.
+, –, *, / and $.

```
stack<double> S;

// scan the input expression from left to right
while (not end of expression)
{
  symb = next symbol in expression;

  if (symb is an operand)
  {
    value = convert symb to double;
    S.push(value);
  }
  else   // symb is an operator
  {
    opnd2 = S.top(); S.pop();
    opnd1 = S.top(); S.pop();
    result = evaluate(symb, opnd1, opnd2);
    S.push(result);
  }
}
final_result = S.top() ;
```

Algorithm to convert an infix expression to postfix format

Observations:

1. The relative order of the operands in postfix format is the same as that in the infix format.
2. If we construct a fully parenthesized infix expression, the postfix expression can be obtained by moving the operators to the corresponding close bracket “)”.
3. In the algorithm, we scan the input infix expression from left to right and use an operator stack to delay the time when the operators are sent to the output postfix expression.

Define the precedence relation of the operators

operators	precedence no.
@	0
‘\0’	1
(	2
+ or –	3
* or /	4
$	5

@ is the special symbol to denote the bottom of stack
‘\0’ denotes the end of input

Defining the precedence relation of the two special symbols ‘\0’ and @ removes the need for special treatment of the boundary conditions.

They are used as sentinels in this algorithm.

```
void infix_to_postfix(char *infix, char *postfix)
{
   stack<char> S;
   S.push('@');
   int i=0, j=0;

   while (S.top() != '\0')
   {
      /* '\0' denotes the end of input; its low precedence
         will cause the other operators to be pop out from
         stack before it is push on to the stack */

     char symb = infix[i++];

     if (isOperand(symb))  // symb >= 'A' && symb <= 'Z'
         postfix[j++] = symb;
     else if (symb == '(')
         S.push('(') ;
     else if (symb == ')')
     {
         while (S.top() != '(')
         {
           postfix[j++] = S.top();
           S.pop();
         } // the ')' is discarded

         S.pop(); // remove '(' from stack
     }
     else if (isOperator(symb))
          //symb is one of {+, -, *, /, $, '\0', '@'}
     {
         while (precNum(S.top()) >= precNum(symb))
         {
            postfix[j++] = S.top();
            S.pop();
         }
         S.push(symb);
     }
   }  //end of while-loop

   postfix[j] = '\0'; //terminate the postfix string
}

//With '@' sitting at the bottom of stack, we need not
//test if stack is not empty before calling top()
```

Implementation of the class stack using array

```
// file: stack.h
#ifndef STACK_H
#defind STACK_H
#include <ostream>

template<class Type>
class stack: public stackADT<Type>
{
private:
   int maxSize;   //var to store the max stack size
   int stackTop;  //var to point to the top element
   Type *list;    //pointer to the array that holds the elements

   void copyStack(const stack<Type>& other);

public:
   stack(int size=100)   //the function body is placed in-line
   {                     //for easy reading
      maxSize = size;
      stackTop = -1;
      list = new Type[maxSize];
   }

   stack(const stack<Type>& other)
   {
      maxSize = 0;
      list = NULL;
      copyStack(other);
   }

   ~stack()
   {  delete [] list;  }

   void initialize();
   {  stackTop = -1;  }

   bool empty() const
   {  return stackTop < 0;  }

   bool full() const;
   {  return stackTop >= maxSize - 1;  }

   int size() const
   {  return stackTop+1;  }
```

```
    const stack<Type>& operator=(const stack<Type>& other)
    {
       if (this != &other)
          copyStack(other);
       return *this;
    }

    void push(const Type& item)
    {
       if (!full())
          list[++stackTop] = item;
       else
          cerr << "Stack overflow" << endl;
    }

    Type& top()
    {
       //precondition: stack is not empty
       return list[stackTop];
    }

    void pop()
    {
       if (!empty())
          stackTop--;
       else
          cerr << "Stack underflow" << endl;
    }

};
```

```
//Implementation of copyStack()
template<class Type>
void stack<Type>::copyStack(const stack<Type>& other)
{
   if (maxSize != other.maxSize)
   {
      if (list != NULL)
         delete [] list;
      maxSize = other.maxSize;
      list = new Type[maxSize];
   }

   stackTop = other.stackTop;
   for (int i = 0; i <= stackTop; i++)
      list[i] = other.list[i];
}

#endif
```

Queue

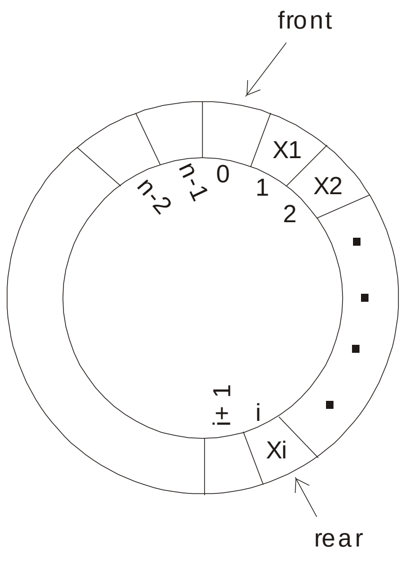
A first-in-first-out (FIFO) queue is an ordered collection of items from which items may be deleted at one end (called the front) and into which items may be inserted at the other end (called the rear).

Operations on a queue :

- initialize: initialize the queue to an empty state
- size: determine the number of elements in the queue
- empty: determine if the queue is empty
- front: retrieve the value of the front element
- back: retrieve the value of the last element (this is not common in the applications of queue)
- push: insert element at the rear of queue (in most textbooks, this operation is called enqueue)
- pop: remove front element (in most textbooks, this operation is called dequeue)

```
template<class Type>
class queueADT
{
public:
   virtual void initialize() = 0;
   //remark: the initialize() function is not part of the
   //        C++ STL. The initialization can be taken care
   //        of by the constructor.

   virtual int size() const = 0;
   virtual bool empty() const = 0;

   virtual Type& front() const = 0;

   virtual void push(const Type& item) = 0;

   virtual void pop() = 0;
}
```

Implementing a FIFO queue using a circular array:



```
Conventions of the representation:
  • The program maintains two indexes, front and rear
  • If the queue is empty, front == rear.
  • If the queue is not empty,
       - rear points to the last element, and
       - front points to the slot before the first element.
```

An array of size *n* can hold up to *n*-1 elements.

Example applications of queue:

- message buffering in inter-process communications
- task scheduling in operating system
- breadth-first search of multi-dimensional data structures, e.g. trees and graphs
- event-driven simulations

```
// file: queue.h
#ifndef QUEUE_H
#defind QUEUE_H
#include <ostream>

template<class Type>
class queue: public queueADT<Type>
{
private:
   int maxSize;
   int queueFront, queueRear;
   Type *list;

   void copyQueue(const queue<Type>& other);

public:
   queue(int size=100)  //the function body is placed in-line
   {                    //for easy reading
      maxSize = size;
      queueFront = queueRear = 0;
      list = new Type[maxSize];
   }

   queue(const queue<Type>& other)
   {
      maxSize = 0;
      list = NULL;
      copyQueue(other);
   }

   ~queue()
   {  delete [] list;  }

   void initialize();
   {  queueFront = queueRear = 0;  }

   bool empty() const
   {  return queueFront == queueRear;  }

   bool full() const;
   {  return (queueRear + 1) % maxSize == queueFront;  }

   int size() const
   {  return (maxSize + queueRear - queueFront) % maxSize;  }
```

```
    const queue<Type>& operator=(const queue<Type>& other)
    {
       if (this != &other)
          copyQueue(other);
       return *this;
    }

    void push(const Type& item)
    {
       if (!full())
       {
          queueRear = (queueRear + 1) % maxSize;
          list[queueRear] = item;
       }
       else
          cerr << "Queue overflow" << endl;
    }

    Type& front()
    {
       //precondition: queue is not empty
       return list[(queueFront + 1) % maxSize];
    }

    void pop()
    {
       if (!empty())
          queueFront = (queueFront + 1) % maxSize;
       else
          cerr << "Queue underflow" << endl;
    }

};
```

```
//Implementation of copyQueue()
template<class Type>
void queue<Type>::copyQueue(const queue<Type>& other)
{
   if (maxSize != other.maxSize)
   {
      if (list != NULL)
         delete [] list;
      maxSize = other.maxSize;
      list = new Type[maxSize];
   }

   queueFront = other.queueFront;
   queueRear = other.queueRear;

   int i = queueFront;
   while (i != queueRear)
   {
         i = (i + 1) % maxSize;
         list[i] = other.list[i];
   }
}

#endif
```

Remark: in the C++ STL, stack and queue are implemented using deque as the default container.

Deque (pronounced like “deck”) is a double-ended queue. It allows insertion and deletion at both ends.

The insertion/deletion functions are called

```
push_front
push_back
pop_front
pop_back
```