EE 441

DATA STRUCTURES WITH C++

LECTURE NOTES

by

UGUR HALICI

CLASSES

ABSTRACT DATA TYPES

ADT: Implementattion independent data description that specifies the contents, structure and legal operations on the data.
An ADT has a

- Name:
- Description of the data structure:
- Operations:
  Construction operations:
    Initial values;
    Initialization processes;
  Other Operations:

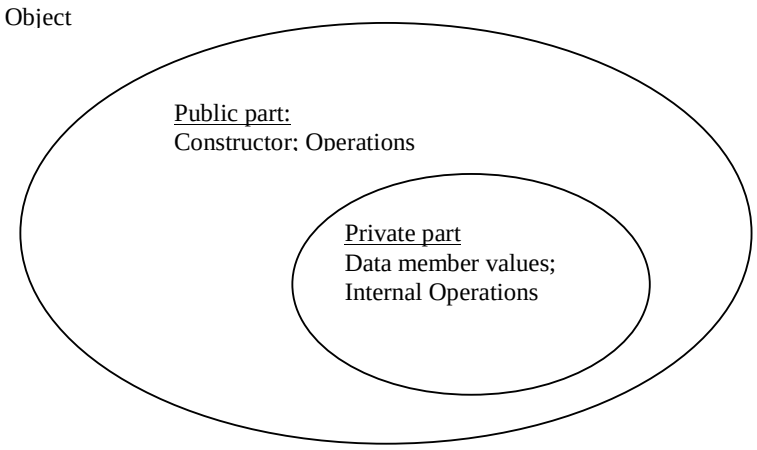
  For each operation:
    Name of the operation;
    Input: external data that comes from the user of this data (client)
    Preconditions: Necesssary state of the system before executing this operation;
    Process: Actions performed by the data
    Output: Data that is output to the client
    Post Conditions: state of the system after executing this operation

Example:

ADT circle
Data
  r≥0 (radius);
Operations
  Costructor:
    Input:radius of circle;
    Preconditions: none;
    Process: Assign initial value of r;
    Output: none;
    Postcondition: none;

  Area:
    Input: none;
    Preconditions: none;
    Process: A ← π*r*r
    Output: A
    Postcondition: none
  Circumference:
    Input: None;
    Preconditions: none;
    Process: C ← 2*π*r
    Output: C
    Postconditions: none;
End ADT circle;

Representation of ADT's in C++



Private: Data and internal operations necessary to implement the class
Public: Operations available to clients (who do not need to know anything about the private parts)

Example:

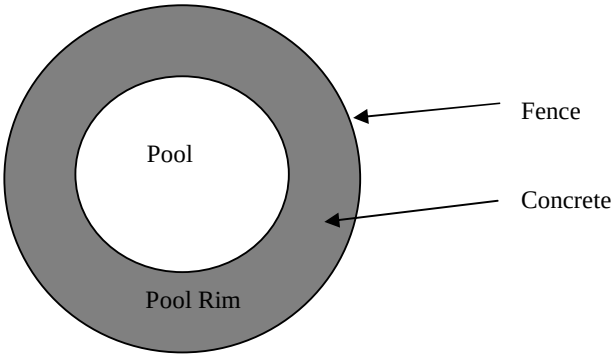
Class Circle
Private: radius (if ist is not to be used anywhere outside the class)
Public: constructor; area; circumference

Clients can only access the public part, i.e., the three methods in this case
Private data members and operations can be acessed only by the methods in the class

An object is an instance of a class type.

The public parts hide information encapsulated in the private parts to:
- Protect data integrity
- Enhance portability
- Facilitate software reuse
- …

Example: Program to compute fence cost and concrete cost for swimming pole




```
# include <iostream.h>  /* necessary for input,outputs */
const float PI=3.141592;
const float FencePrice=0.5; /*MTL per meter */
const float ConcretePrice=0.8 /*MTL per squaremeter ?/
/ * Class decleration */
class Circle
{ private:
    float radius;
  public:
    Circle(float r); /* constructor function */
    float Circumference(void) const; // this is  a constant function since the object
                                     // is not changed when it is called
    float Area(void) const;
}
/* Class imlementation */
/* constructer */
Circle::Circle(float r)
{
    radius=r
}
/* return circumference */
float Circle::Circumference (void) const
{
    return  2*PI*radius
}
/* return area */
float Circle::Area(void) const
{
    return PI*radius*radius
}


void main()  /* Now let's use this data structure
```

```
{ float radius;  /* pool radius,
                 /* this has the same name with the encapsulated private data */
                 /* but does not cause any problem */
  float FenceCost, ConcreteCost;

/* arrange output format to 2 decimal place */
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision (2);


/* get radius from user */
    cout << "Enter pool radius>";
    cin >> radius;
/* create two different instances off class Circle, i.e. Circle objects */
    Circle Pool(radius);
    Circle PoolRim(radius+3);

/* Note: we did not have to say anything about centers*/
/* compute fence cost and output it */
    FenceCost=PoolRim.Circumference()*FencePrice;
    Cout<<"Fence cost is:" <<FenceCost<<"YTL"<<endl;
/* ..and compute concretecost and output it */
    ConcreteCost=(PoolRim.Area()-Pool.AreA())*ConcretePrice;
    Cout<<"Concrete cost is: "<<ConcreteCost<<"YTL<<endl
}
/* sample run:

Enter pool radius> 5
Fence cost is: 15.70  MTL
ConcreteCost is: 62.83 MTL
*/
```

CLASS DECLARATION

```
Class   <ClassName>
{
private:
        <private data declarations>
        <private method declarations>
public:
        <public data declarations>
        <public method declarations>
}
```

Example:

```
Class Rectangle
{
private:
float length, with;
        // Note: length and width are data members of ADT ; by placing data members
        // in private part, we ensure that only defined methods may change them.
public:
Rectangle (float l=0, float w=0); // constructor
        // Note: constructor has the same name as the class
        // methods to retrieve and modify private data GetLength, PutLength, //GetWidth,
        Putwidth
float GetLength(void) const;
        // Note:const indicates that the method is  constant, therefore no class data
        // item may be modified, i.e. GetLength does not change the state of
        // the Rectangle object
void PutLength (float l);
float GetWidth(void) const;
void PutWidth(float w);
float Perimeter(void) const;
float Area(void) const;
};
```

OBJECT DECLARATION:

Object declaration creates an instance of a class:

<classname> <object(parameters)>;

```
Rectangle     room(12,10);
Rectangle t; // omitting parameters means using default values, i.e. l=0, w=0
Rectangle square(10,10), yard=square, S;
        // yard is initially identical to square, S is created with default length, with=0.
```

Public members of an object can be accessed :

```
x=room.Area(); // assigns  Area=12*10=120
t.PutLength(20); //assigns 20 as length of Rectangle t

cout<<t.GetWidth(); // outputs current value of width of t, which is 0
cin>>x;
t.PutLength(x); // inputs x and assigns it to length of t.
```

If a method is not defined inside the class declaration (i.e.if not defined inline), it must be defined outside

<ReturnValueType> <ClassName>::<FuctionName(parameters)>

```
float Rectangle::GetLength(void) const
// using :: makes function belong to class, so it can access private members
{ return Length; // access private member length
}
```

the same result may be achieved as:

```
Class Rectangle
{
Private:
        float length, width;
Public:
        ….
        float Get Length(void) const  // code is given inline
        {
            return Length;
        }
        ….
}
```

Note: constructor and other methods can be defined inline or outside the class body.

ALTERNATIVE CONSTRUCTORS

More than one constructer can be defined for a class

Example:

```
# include  <string.h>
# include  <string.h>
class Date
{
private:
        int month, day, year;
public:
        Date (int m=1, int d=1, int y=2000);
        Date(char *dstr);
// Note: two different constructors are defined; The compiler will select the appropriate
// constructor according to the call parameters during object creation
        void PrintDate(void); // the only method other than the constructor
};

// Now the methods will be defined externally:
Date:.date(int m, int d, int y): month(m), day(d), year(y)
{ }
// Alternative constructor
Date::Date(char *dstr)
```

```
{       ……
        // Here some operations are needed to read string in the form “dd/mm/yyyy”
        // from input stream, and then convert it to integer month, day, year
}

//Now the print method
void Date::PrintDate(void)
{       //static table of months
        static char *Months[]={"","january", …,"December"};
        cout<<Months[month]<<" "<<day<<", "<<year;
}
```

Assuming the class date is contained in the file "date.h"

```
#include <şiostream.h>
#include "date.h"
void main(void)
{
        date day1(10,29,1923);
        date day2;
        date day3("07/10/2007");
        //note that,  mm/dd/yyyy will be replaced with today's date in class
        cout<<"The Turkish Republic was founded on";
        day1.printDate(); cout<<endl;
        cout<<"The first day in the 21st century was";
        day2.Printdate();cout<<endl;
        cout<<"Today is";
        day3.PrintDate();cout<<endl;
}
```

TEMPLATES

We may want to use the same class or function definition for different types of items, i.e. we may want to create different objects within the same class, but whose datatypes are different or we may want to define a general function that works on different datatypes. It would be nice if we could define the data type with the object or with the function call.

Example

```
Stack <float> A;
Stack<char> B;
SeqList<int> C;
SeqList<char> D; etc.
```

C++ allows this through the usage of templates

```
template <class T1, Class T2, …ClassTn>
```

indicates that T1,T2, ..Tn are classess that will be used with a specific class upon creation of an object

```
template <class T>
// Note here that any operation in the template class or function must be
//defined for any possible data types in the template
int SeqSearch(T list[ ], int n, T key)
// T is a type that will be specified when SeqSearch is called
{
   for (int i=0; i<n; i++)
        if list[i]==key)
                return i;
   return -1;
}
```

Now call SeqSearch with different datatypes:

```
int A[10], Aindex, Mindex
float M[10], fkey=4.5;
Aindex=SeqSearch(A,10,25); //search for int 25 in A
Mindex=SeqSearch(M,100,fkey); //search for float 4.5 in M
```

C++ OPERATION ON ARRAYS

Decleration examples:

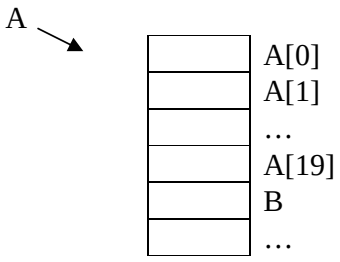
```
Const int ArraySize=50;
        float    A[ArraySize];
        long     X[ArraySize+10];
```

assignment examples:

```
A[i]=z;
t=X[i]
X[i]=X[i+1]=t /* this is equivalent to X[i+1]=t;  X[i]=X[i+1] , i.e. right to left */
```

Example:

```
int V=20;
int A[20]; /*  index range is 0-19  */
// A is a pointer  contains the starting address of the array, same as the address of A[0]
int B;
```




```
A[V]=0; /* index is out of range, but most C++ compilers don't check this */
```

Effect is B=0, since the first location after A is reserved for B by above decleration

TWO DIMENSIONAL ARRAYS

Example:

```
int T[3] [4]={{20,5,-3,0},{-85,35,-1,2},{15,3,-2,-6}};
```

T:

20	5	-3	0	← T[0]
-85	35	-1	2	← T[1]
15	3	-2	-6	← T[2]

```
a=T[2] [3]; T[0][4]=a;
```

Example:

```
int T[ ][5] // list of 5 element arrays
```

ARRAYS OF OBJECTS

Rectangle room[100]; // constructor is called for room[0] .. room[99];

This declaration creates an array of 100 objects. Each have different members, all initialized in this case to the default values.

Note: For declaring large array objects, a constructor with default values or with no parameters is preferred.

Rectangle room[3]={rectangle(10,15), Rectangle(5,8), Rectangle(2,30)}

may be practical, but

Rectangle room[100];

simply initializes all length and with values to the default value of 0.

Notes:

```
void f(char X[ ])
{… }
```

is equivalent to

```
void f(char *X)
{… }
```

when calling

```
void main ()
{
   Char A[100]
   ….
   f(A)  // here A is pointer to array
   ….
}
```

ARGUMENT PASSING, RECURSIVE FUNCTIONS, COMPLEXITY OF ALGORITHMS

ARGUMENT PASSING

Suppose that we desire to swap the contents of two integer variables. Consider the following function:

```
void swap(int v1, int v2)
{ int tmp=v2;
  v2=v1;
  v1=tmp;
}

main()
{ int i=10;
  int j=20;
  cout << "Before swap():\ti:"<<i<<"\tj:"<<j<<endl;
  swap(i,j);
cout << "After swap():\ti:"<<i<<"\tj:"<<j<<endl;
}
```

when executed the result is not as we desired, but it is

```
Before swap(): i:10 j:20
After swap():  i:10  j:20
```

Two alternatives to solve the problem:

The first alternative is to use pointers as parameters:

```
void  pswap(int *v1, int *v2) // parameters are pointers
{  int tmp=*v2;
   *v2=v1;
   *v1=tmp;
}

main ()
{.....
pswap(&i, &j); // send address of i and j as parameter
...
}
```

When executed, we will have:

```
Before swap(): i:10 j:20
After swap():  i:20  j:10
```

Second alternative is to use reference:

```
void rswap(int &v1, int &v2)
{ int tmp=v2;
      v2=v1;
      v1=tmp;
}


main ()
{.....
rswap(i, j);
...
}
```

When compiled and executed, we will have

```
Before swap(): i:10 j:20
After swap():   i:20  j:10
```

However if it is declared as

```
void crswap(const int &v1, const int &v2)
{ int tmp=v2;
      v2=v1;
      v1=tmp;
}
main ()
{.....
crswap(i, j);
...
}
```

When executed, we will have

```
Before swap(): i:10 j:20
After swap():   i:120  j:20
```

Pointer Types:

A pointer variable holds values that are the addresses of objects in memory. Through a pointer an object can be referenced directly. Typical uses of pointers are the creation of linked lists and the management of objects allocated during execution.

```
int* ip // pointer decleration
int *ip // the same as above
int *ip1, *ip2 // two pointers
int* ip1, ip2 // a pointer, an integer
```

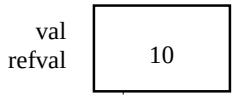
```
int ip, *ip2 // an integer and an integer pointer
long *lp, lp2 // a long integer pointer and a long integer
float fp, *fp2 // a floating point and a floating point pointer

int i=1024
//create an  pointer that points to i
int *ip=i // error, type mismatch
int *ip=&i // ok. the operator & is referred as address-of operator
// create another pointer that also points to i
int *ip2=ip // ok. now ip2 also addresses i
// to create a pointer that points  ip2
int *ip3=&ip // error, type mismatch
int **ip3=&ip2 // ok , it is a pointer to a pointer

int i=1024
int *ip=&i; //ip points to i
int k=ip; //error
int k=*ip; // k now contains 1024

int *ip=… //  type decleration for the pointer and initialization
*ip= … // the item pointed by ip
…=*ip // the item pointed by ip
   =ip // the pointer itself

int i
int k=-5
int *ip=&i;
*ip=k; // i=k
*ip=abs( *ip) // i=abs(i);
ip=ip+1
// ip points to an integer that is 4 bytes. The value of the address it contains is increased
// by the size of the object.,i.e. it is incremented by sizeof(int)=4
// However , some compilers needs it is to be written explicitly as  ip+1*sizeof(int)
int i,j,k;
int *ip=&i;
*ip= *ip+2 // add two to i, that is  i=i+2
ip=ip+2; // now it  contains ip+2* sizeof(int)
```

Reference Types:

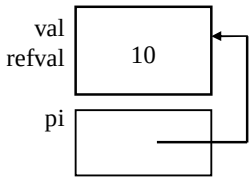
All the operations applied to the reference act on the object to which it refers

```
int val=l0
int  &refval=val
```

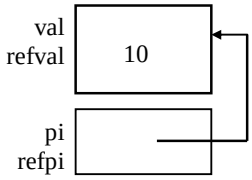
demonstrate reference as



int *pi=&refval //initializes pi with the address of val



int *&refpi=pi // refpi is a reference to a pointer



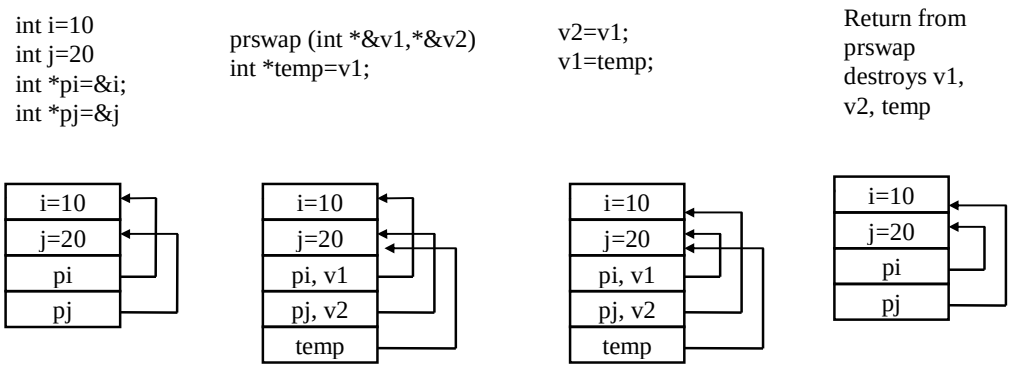
int &*ptr=&refval //USELESS
// ptr is a pointer to a reference,
//however the address of a reference is same as the referenced variable
//therefore instead &*ptr simply use *ptr as below
int *ptr=&refval
// or
*ptr=&val
)

A pointer argument can also be declared as a reference when the programmer wish to modify the pointer itself rather than the object addressed by the pointer

```
void prswap(int *&v1, int *&v2)
//NOTE:  *&v1 should be read from right to left  v1 is a reference to a pointer to
// an  object of type int
{ int tmp=v2;
      v2=v1;
      v1=tmp;
}
main()
{ int i=10;
  int j=20;
  int *pi=&i;
  int *pj=&j;

  cout << "Before swap(): i:"<<*pi<<"j:"<<*pj<<endl;
  prswap(pi,pj);
  cout << "After swap():  i:"<<*pi<<"j:"<<*pj<<endl;
}
```

When compiled and executed, we will have

```
Before swap(): *pi:10 *pj:20
After swap():   *pi:120  *pj:20
```



before swap *pi: 10 *pj: 20
after swap *pi: 20 *pj:10

Notes:

const int* p;

p=… OK
*p= not OK

int * const p;

p=… not OK
*p=… OK

const int* const p;

p=… not OK
*p=… not OK

const int &p
p=… not OK

int & const p;
// useless, a reference is already a constant, you can not change shortcut

RECURSIVE FUNCTIONS.

Some calculations have recursive character: for example N!=N*(N-1)!. To be able implement this kind of functions we write functions that call themselves. Such functions are called recursive functions.

Example:

```
int factorial (int n)
{if n>1
     return n*factorial(n-1)
else
    return 1
}
```

Example: Quick Sort is an algorithm for sorting numbers in a recursive way.

elem											
5	3	9	17	2	20	13	1	4	6	7	
lo→	lo→	lo						←hi	←hi	←hi	←hi
		4						9			
		lo→	lo				hi	←hi			
			1				17				
			lo→	lo→	lo						
				hi	←hi	←hi	←hi				
2				5							
2	3	4	1	5	20	13	17	9	6	7	

```
Template <classType>
void qsort(Type *ia, int low, int high)
{//quick sort
     if (low<high) {
         int lo=low;
         int hi=high+1;
         Type elem=ia[ low]
         while (True)
        {
           while (ia[ ++lo]<elem);
              while (ia[ --hi]>elem);
           if (lo<hi)
                    swap(ia,lo,hi);
           else break;
         }
        swap(ia,low,hi);
        qsort(ia,low,hi-1);
        qsort(ia, hi+1,high);
    }
}
```

Quicksort makes use of a helping function swap (). This, too, needs to be defined as a template function:

```
template <class Type>
static void swap (Type ia[], int i, int j)
{// swap two elements of an array
  Type tmp=ia[i];
       ia[i]=ia[j]
       ia[j]=tmp
}
```

or it can be implemented as

```
template <class Type>
static void swap (Type *ia, int i, int j)
{// swap two elements of an array
  Type tmp=*(ia+i);
       *(ia+i)=*(ia+j)
       *(ia+j)=tmp
}
```

```
#include "qsort.c"
main(){
int       A[11]={5,3,9,17,18,20,13,2,4,6,7}
double  B[5]={12.8,76.0,87.7,98.6, 65.7}
main ()
qsort(A,0,10)
qsort(B,0,4)
}
```

COMPLEXITY OF ALGORITHMS

Search an n-element array for a match with a given key; return the array index of the matching element or -1 if no match is found

```
Int SeqSearch(DataType list[ ], int n, DataType key)
// Note dataType must be defined earlier
// e.g. typedef int DataType, or  typedef  float DataType etc.
{
        for (int i=0; i<n; i++)
                if (list[i]==key)
                        return i;
        return –1;
}
```

In the worst case n comparisons are performed. Expected (average) number of comparisons is n/2 and expected computation time is proportional to n. That is, if the algorithm takes 1 msec with 100 elements, it takes ~5msec with 500 elements, ~200 msec with 20000 elements, etc.

Example: Binary Search Algorithm (using a sorted array)

```
int BinarySearch(DataType list[ ], int low, int high, DataType key)
{
int mid;
DataType midvalue;
   while (low<=high)
   {
        mid=(low+high/2; // note integer division, middle of array
        midvalue=list[mid];
        if (key==midvalue) return mid;
        else     if (key<midvalue) high=mid-1;
                 else low=mid+1;
   }
return –1
}
```

HW:Write a ecursive function for binary search

Example :

```
int list[5]={5,17,36,37,45}

Binary Search(list, 0,4,44}

1          mid=(0+4)/2=2
           midvalue=17
           key>midvalue
           low=3
```

2 mid=(3+4)/2=3
midvalue=37
key>midvalue
low=4

3 mid=(4+4)/2=4
midvalue=45
key<midvalue
high=3

4 since high=3<low=4, exit
return –1 (not found)

Binary Search(list, 0,4,5}

1 mid=(0+4)/2=2
midvalue=17
key<midvalue
high=1

2 mid=(0+1)/2=0
midvalue=37
midvalue=5
key=midvalue
return 0 (found)

In the worst case Binary search makes $\lceil \log_2 n \rceil$ comparisons

Example:

n	8	20	32	100	128	1024	65536
$\lceil \log_2 \rceil$	3	5	5	7	7	10	16

Assume Binary Search 1 msec with 100 elements, then it takes 7 msec with 12800 elements, 16 msec with 6553600 elements

n	Sequential Search O(n)	Binary Search O(log n)
100	1 msec	1 msec
12800	128 msec	7 msec
6553600	6553600 msec	16 msec

Asymptotic upper bound: Big-Oh

Function f(n) is O(g(n)) if there exists a constant K and some n_0 such that $f(n) \leq K*g(n)$ for any $n>n_0$. That is as $n \rightarrow \infty$, f(n) is upper bounded by a constant times g(n).

Asymptotic lower bound: Omega

Function f(n) is $\Omega(g(n))$ if there exists a constant K and some n_0 such that $f(n) \geq K*g(n)$ for any $n>n_0$. That is as $n \rightarrow \infty$, f(n) is lower bounded by a constant times g(n).

Asymptotic upper- lower bound : Theta

f(n) is $\theta(g(n))$ iff f(n) is O(g(n)) and $\Omega(g(n))$

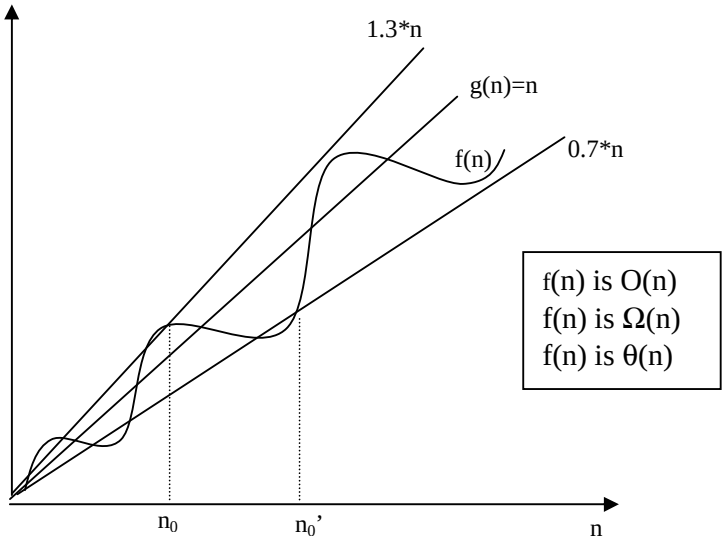
Usually g(n) selected among
log n (note $\log_a n = k*\log_b n$ for any a and b)
n, n^k (polynomial)
k^n (exponential)

Use O(.) to show how good is an algorithm
For example $O(n^2)$, polynomial, feasible

Use Ω(.) to show how bad it is
for example $\Omega(2^n)$, grows exponentially, not feasible

Example: $f(n)= n^3+70n^2+2$ is $O(n^3)$, and also $\Omega(n^3)$, $\theta(n^3)$
Example: $f(n)=2^n+10^{23}n+n^{1000}$ is $\Omega(2^n)$, and also $O(2^n)$, $\theta(2^n)$

Example



Example: $\sum_{i=1}^{n} i$

	int Sum (int n)	
	{	
1	int result = 0;	t1
2	for (int i = 1; i <= n; ++i)	t2a, t2b,t2c
3	result += i;	t3
4	return result;	t4
	}	

t1+ t2a + (n+1) t2b + n*t2c +n*t3+t4= tA+n*tB
$\Rightarrow$O(n), Ω(n), θ(n)

Example $\sum_{i=0}^{n} a_i x^i$

	int (int a [], int n, int x)	
	{	
1	int xpower=1	t1
2	result = a [0]*xpower;	t2
3	for (int i = 1; i <= n; i++);	t3a, t3b, t3c
4	xpower=x*xpower;	t4
5	result = result+ a [i]*xpower;	t5
6	return result;	t6
	}	

t1+t2+t3a+(n+1)*t3b+n*(t3c+t4+t5)+t6=tA+n*tB $\Rightarrow$ O(n), Ω(n), θ(n)

Example: $\sum_{i=1}^{n}\sum_{j=1}^{i} i*j$

	int DSum (int n) {	
1	int result = 0;	t1
2	for (int i = 1; i <= n; ++i)	t2a, t2b,t2c
3	for (int j=1;j<=i,++j)	t3a, t3b, t3c
4	result += i*j;	t4
5	return result;	t5
	}	

$t1+ t2a + (n+1)\ t2b + n*t2c +$

$\sum_{i=1}^{n}(t3a + (i+1)*t3b + i*t3c + i*t4)$

$+t5$

$= t1+ t2a+t2b+t5+n*(t2b+t2c+t3a+t3b)+ (n*(n+1)/2)*(t3b+t3c+t4)$

$= tA+n*tB+n^2*tC \Rightarrow O(n^2), O(n^2), \theta(n^2)$

Example: Recursive program to compute n!

T(n) {
int Factorial (int n)
{
if (n = = 0) } tB
return 1; }
else
return n * Factorial (n - 1); } T(n-1)+tA
}

$$T(n) = \begin{cases} tB & n = 0 \\ tA + T(n-1) & n > 0 \end{cases}$$

$T(n)= T(n-1)+ tA = T(n-2) + 2tA=\ldots=T(0)+n*tA=tB+n*tA \Rightarrow O(n)$

Example:

assume myfunc1: $\theta(n)$

assume myfunc2: $\theta(n^2)$

```
int RandomFunc(int n, int seed)
{
    int x=Rand(seed); //assume Rand function returns a random integer
    for (int i=1; i<=n; i++)
        if (x%2== 0) //if x is even
            x=Rand(x)+myfunc1(x,n)        θ(1)+ θ (n) = θ (n)
        else
            x=Rand(x)+myfunc2(x,n)        θ(1)+ θ (n^2) = θ (n^2)
     return x
}
```

repeated n times

Upperbound

$O(1)+n*(\max(O(n),O(n^2))+O(1)=O(n^3)$

Lowerbound: take the minimum

$\Omega(1)+n*(\min(\Omega(n), \Omega(n^2))+ \Omega(1)= \Omega(n^2)$

Example: compute x^n recursively

```
int Power (int x, int n)
{
   if (n == 0)
      return 1;
   else if (n % 2 == 0) // n is even
      return Power (x * x, n / 2);
   else // n is odd
      return x * Power (x * x, n / 2);
}
```

$$T(n) = \begin{cases} ta, & n = 0 \\ tb + T(\lfloor n/2 \rfloor), & n > 0, \text{ n is even} \\ tc + T(\lfloor n/2 \rfloor), & n > 0, \text{ n is odd} \end{cases}$$

where tb<tc

As the first attempt to solving recurrence, Let's suppose that $n=2^k$, so n is even

$T(2^k) = tb + T(2^{k-1}) = 2tb + T(2^{k-2})=..=k*tb+T(2^0)=k*tb+tc+T(0)=k*tb+tc+ta$

All even exept the last two steps (best case)

$k=\log_2 n \Rightarrow T(n)=(\log_2 n)*tb+tc+ta \Rightarrow \Omega(\log_2 n)$

Now consider the worst case : suppose $n=2^k-1$

For example n=31, the program calls itself recursively to compute x^{15}, x^7, x^3, x^1 and x^0, all except the last one are odd powers of x

For $n=2^k-1$

$T(2^k-1)= tc + T(2^{k-1}-1) = 2*tc + T(2^{k-2}-1)=..=k*tc+ T(2^0-1)=k*tc+T(0)=k*tc + ta$

This time, $k= \log_2(n+1) \Rightarrow T(n)=(\log2(n+1))*tc+ta$
$< (\lfloor \log_2 n \rfloor+1)*tc+ta \Rightarrow O(\log_2 n)$

Therefore the algorithm is $\theta(\log_2 n)$

Fibonacci numbers:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

i.e. 0,1,1,2,3,4,8,13,21,34…

Example: nonrecursive program to compute Fibonacci numbers

```
int Fibonacci (int n)
{
   int previous = -1;
   int result = 1;
   for (int i = 0; i <= n; ++i)
   {
      int const sum = result + previous;
      previous = result;
      result = sum;
   }
   return result;
}
```

Complexity O(n), Ω(n), θ(n)

Example: Recursive program to compute Fibonacci numbers

```
int Fibonacci (int n)
{
   if (n = = 0 || n = = 1)
      return n;
   else
      return Fibonacci (n - 1) + Fibonacci (n - 2);
}
```

$$T(n) = \begin{cases} \theta(1) & n < 2 \\ T(n-1) + T(n-2) + \theta(1) & n \geq 2 \end{cases}$$

T(1) = θ(1)
T(2) = θ (1)
T(3) = θ (1)*(1+1)
T(n) = T(n-1) + T(n-2)

T(n)= θ (1)*Fib(n) = θ (Fib(n))
// It's complexity is also Fibonacci
// grows very rapidly

$$T(n) = \begin{cases} \theta(1) & n < 2 \\ T(n-1) + T(n-2) + \theta(1) & n \geq 2 \end{cases}$$

Notice that $T(n)=T(n-1)+T(n-2) + \theta(1) \leq 2T(n-1) \leq 2^2 T(n-2) +2\theta(1)$ ….
$\leq 2^{n-2}T(2) +(n-2)\ \theta(1) = 2^{n-2}\theta(1)+(n-2)\theta(1)$
$\Rightarrow T(n)$ is $O(2^n)$

This time it is better to indicate a lower bound rather than an upper bound, i.e.$\Omega(.)$

Case: n is even

$T(n)= T(n-1)+T(n-2) +\theta(1) \geq 2(T(n-2)) \geq 2^2 T(n-4)\ldots \geq 2^{(n-2)/2}T(2) \Rightarrow \Omega(2^{n/2})$

Case: n is odd
$T(n)= T(n-1)+T(n-2) +\theta(1) \geq 2(T(n-2)) \geq 2^2 T(n-4)\ldots \geq 2^{(n-1)/2}T(1) \Rightarrow \Omega(2^{n/2})$

So $T(n)$ is $\Omega(2^{n/2})$

i.e. Exponential, so infeasible

In fact Fib(n) is $\Omega((3/2)^n)$

STACKS AND QUEUES

STACKS

A stack is a data structure consisting of a list of items and a pointer to the "top" item in the list. Items can be inserted or removed from the list only at the top, i.e. the list is ordered in the sequence of entry of items, ans insertions and removals proceed in the "LIFO" last-in-first-out order.

```
# include <iostream.h>
# include <stdlib.h>
const int MaxStackSize=50;
template <class T>
Class stack
{
 private:
   T stacklist[MaxStackSize],
     int top;
 public:
     Stack(void); // constructor to initialize top
 //modification operations
     void Push(const T& item);
     T Pop(void);
     void Clearstack(void);
//just copy top item without modifying stack contents
     T  Peek(void) const;
//check stack state
     int StackEmpty(void) const;
     int StackFull(void) const;
}

// Now implementation of Stack methods
     template <class T>
     Stack<T>::Stack(void):top(-1)
     {}
//Stack Empty
int stack:.StackEmpty(void) const
{
  return top==-1; //value is 1 if equal, 0 otherwise
}
// Stack Full
int Stack::StackFull(void)const
{
 return topMaxstackSize-1;
}

//Push
```

```
template <class T>
void Stack<T>::Push(const T& item)
{
//can not push if stack has exceeded its limits
    if  StackFull( )
         {
          cerr<<"Stack overflow"<<endl;
          exit(1);
          }
// increment top ptr and copy item into list
    top++;
    stacklist[top]=item;
}

//pop
template <class T>
T Stack<T>::Pop(void)
    {
      T temp;
      // is stack empty nothing to pop
      if StackEmpty( )
      { cerr<<"Stack empty"<<endl;
           exit(1);
        }
    //record the top element
           temp=stacklist[top];
//decrement top and return the earlier top element
           top--;
           return temp;
   }
//Peek is the same as Pop, except top is not moved
template <class T>
    T Stack:.Peek(void) const
    {// write Peek as exercise
    }
//Clear Stack
void Stack::ClearStack(void)
{
 top=-1;
}
```

Example:
Write a program that uses a stack to test a given character string and decides if the string is a palindrome (i.e. reads the same backwards and forwards, e.g. "kabak", " a man a plan a canal panama", etc.)

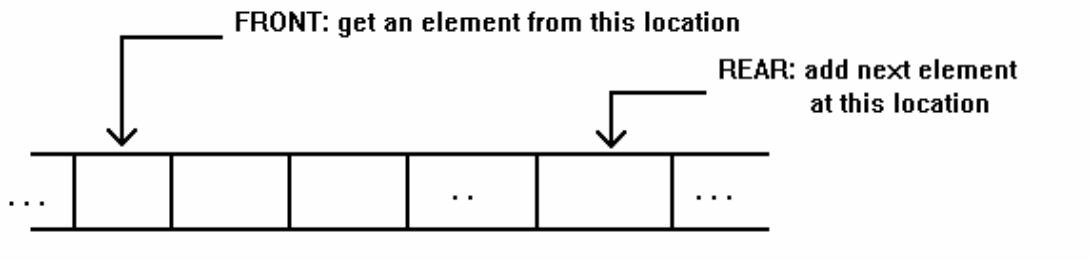
```
// Algorithm:
// first get rid of all blanks in the string, then push the whole string
// characterwise, into a stack, then pop out characters one by one,
// comparing with the characters of the original de-blanked string
// Assuming that the stack declaration and implementation  are in "stack.h"
# include "astack.h"
#include <iostream.h>
void Deblank(char *s, char *t)
{while (*s!=NULL)
   {if (*s!='  ')
        *t++=*s;
    s++;
   }
*t=NULL; //append NULL to newstring
}

void main ()
{const int True=1, False=0;
// create stack object to store string in reverse order.
Stack s;
char palstring[80], deblankedstring[80], c;
int I=0; // string pointer
int ispalindrome=True; //we'll stop if false
// get input
    cin. Getline(palstring,80,'\n');
//remove blanks
    Deblank(palstring,deblankedstring);
//push character onto stack
    i=0;
    while(deblankedstring[i]!=0)
    {
      S.Push(deblankedstring[i]);
      i++;
   }
//now pop one-by-one comparing with original
    i=0;
    while (!s.StackEmpty())
{
    c=s.Pop();
    //get out of loop when first nonmatch
    if (c!=deblankedstringi)
    {isPalindrome=False;
    break;
    }
// continue till end of string
```

```
    i++;
}
//operation finished. Printout result
if (ispalindrome)
    cout<<"\"<<palstring<<"\"<<"is a palindrome<<endl;
else
    cout<<"\"<<palstring<<"\"<<"is not a palindrome<<endl;
```

QUEUES

A queue is a data structure that stores elements in a list and permits data access only at the two ends of the list. An element is inserted at the rear of the list and deleted from the front of the list.

Elements are removed in the same order in which they are stored and hence a queue provides FIFO(first-in/first-out) or FCFC(first-come/first-served) ordering



	Operation
A Fro Rear nt	Arrival of A
A B Front Rear	Arrival of B
A B C Front Rear	Arrival of C
B C Front Rear	Departure of A
C Front Rear	Departure of B

Linear Queues

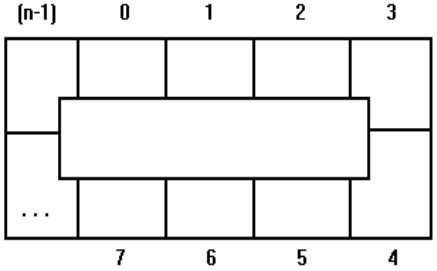
Consider the following operations performed on a queue of size four. The values of pointers front and rear, and the contents of the queue are shown below after each operation.

operation	front	rear	1	2	3	4	element	
	1	1						Queue is empty
1. add 12	1	2	12				12	
2. add 15	1	3	12	15			15	
3. add 17	1	4	12	15	17		17	
4. delete	2	4		15	17		12	
5. delete	3	4			17		15	
6. add 10	3	5			17	10	10	Queue is full
7. add 5	3	5			17	10		Error

Although there is more space, it can not be used, so a better representation is needed.

Circular Queues

For a more efficient queue representation, consider the array to be circular:



Initially FRONT=REAR=COUNT=0,

There are n locations in the queue, 0,1,...(n-1), and count=0 corresponds to the case the queue is empty. Initially front=rear=0.

We insert a new element at the location pointed by rear, and then we update the rear pointer to point to the next available location. As we insert elements, rear becomes rear+1, however if we insert an element when rear=(n-1), rear becomes 0. The front pointer changes similarly.

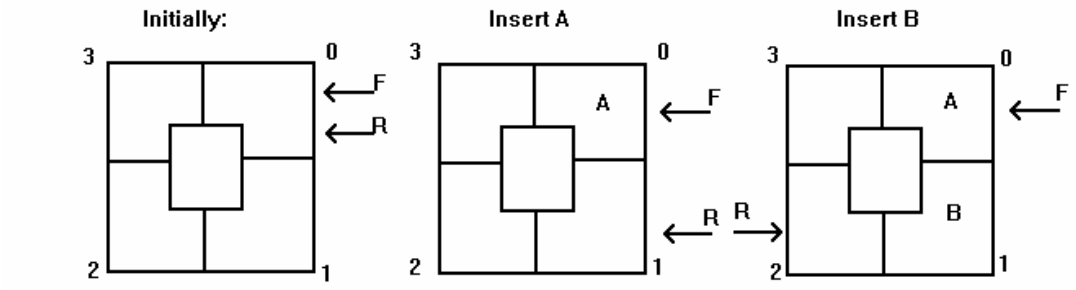
Now consider the case front=0 and rear=(n-1), count=(n-1), that is there are (n-1) elements in the queue. If we insert another element, queue becomes full and rear=front=0 and count=n. Therefore in both cases whether the queue is empty or full we have rear=front . The cases the queue empty or full are discrimnated by the value of count.

One way of overcoming this problem is using only (n-1) locations of the queue. Then, rear=front represents the case the queue is empty, and

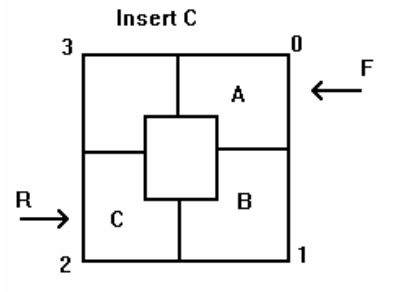
((rear+1) mod n)=front corresponds to the case the queue is full.

Another method might be keeping a flag to indicate whether the queue may become full or empty when rear becomes equal to front.

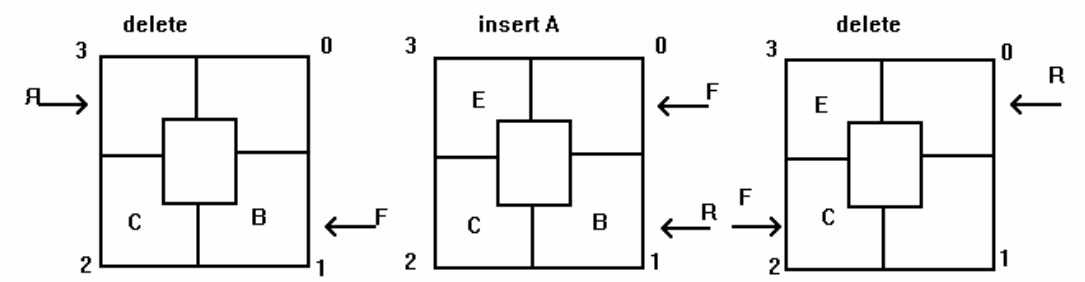
Example:

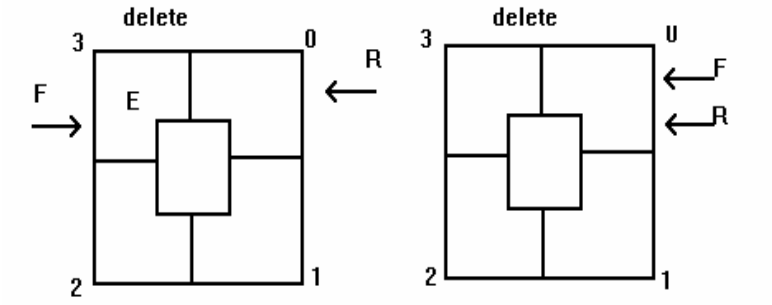


F=0, R=0, C=0 F=0, R=1, C=1 F=0, R=2,C=2



F=0, R=3, C=3





Implementation of Queue in C++

Decleration:

```
#include <iostream.h>
#include <stdlib.h>
```

```
const int MaxQSize=50;
template <class DataType>
class Queue
{private:
 // queue array and its parameters
int front, rear, count
DataType qlist[MaxQSize];
public:
//constructor
Queue(void); // initialize data members

//queue modification operations
void Qinsert(const Datatype item);
DataType Qdelete(void);
void ClearQueue(void);

// queue access
DataType qFront(void) const;

// queue test methods
int QLength(void) const;
int QEmpty(void) const;
int Qfull(void) const;
};
```

Implementation:

```
// Queue constructor
//initialize queue front, rear, count
Queue::Queue(void): front(0), rear (0), count(0)
{}


// queue test methods
int Queue::QLength(void) const
{return count};
int Queue::QEmpty(void) const
{return (count= =0)};
int Queue::QFull(void) const
{return (count= =MaxQSize) };

//Queue Modification Operations

//Qinsert: insert item into the queue
template <class DataType>
void Queue::Qinsert(const Datatype& item)
{
// terminate if queue is full
   if QFull( )
   {
```

```
     cerr<<"Queue overflow! <<endl;
     exit(1)
    }
//increment count, assign item to qlist and update rear
    count++;
    qlist[rear]=item;
    rear=(rear+1)% MaxQSize;

//QDelete : delete element from the front of the queue and return its value
template <class DataType>
Datatype Queue::QDelete(void)
{
  DataType temp;
  // if qlist is empty, terminate the program
  if QEmpty( )
  {
     cerr<<"Deleting from an empty queue!"<<endl;
  }
  //record value at the front of the queue
 temp=qlist[front];
//decrement count, advance front and return former front
 count --;
 front=(front+1) % MaxQsize;
 return temp;
}

// queue access
template <class DataType>
DataType Queue::qFront(void) const
{return qlist[front]};
```