# Benchmark Design Concepts

# Contents

- Basic concepts of benchmarking
- Processor benchmarks
- Disk benchmarks

# Conditions that benchmark programs must satisfy

- Reliable work in all environments
- Same workload in all environments
- Reproducibility of results
- No machine dependent components

# Reliable work in various environments

- Use algorithms that are sufficiently tested, reliable and generate stable results
- Benchmark programs must satisfy highest portability standards
- Benchmark programs should perform same operations in all hardware and software environments

# Stability of workload

- Benchmark workload must be the same for all competitive systems:
  - All systems must use the same algorithms
  - All systems must use the same data
- Benchmark programs should not use machine-dependent or OS-dependent components

# Reproducibility of results

- Benchmark programs are regularly executed multiple times to compute the average run time, or to adjust desired run time
- In each execution benchmark program must generate same results, i.e. results must be reproducible

# Compilation of benchmark programs

- Avoid using different compilers for different computer systems or different operating systems.
- Make sure to always compile benchmark programs using the same (highest) level of optimization
- Avoid using debug versions of programs for benchmarking

# Selection of benchmark workloads

- In the case of specific natural workload that benchmarks must simulate, benchmark workload is selected to be as similar to the natural workload as possible
- If information about natural workload is not available, we use default workload that includes operations that are frequent in all cases of data processing

# Default workloads

- Default processor workload contains two groups of operations:
  - Floating-point operations
  - Integer and combinatorial operations
- Default disk workload includes two groups of operations
  - Sequential write/read of a disk file
  - Random write/read of a disk file

# Speed indicators

- If n benchmark programs $B1,\ldots,Bn$ run for $T1,\ldots,Tn$ seconds, then the computer speed in operations per minute can be defined as $V1=60/T1,\ldots, Vn=60/Tn$
- The overall (average) speed indicator V is computed as the harmonic mean of individual speed indicators:

$$V = n/(1/V1 + \ldots +1/Vn)$$
$$= 60n/(T1 + \ldots + Tn)$$

# Processor benchmarks

# The size of benchmark

- If a benchmark program is large and/or processes large data sets, then it will use processor, cache memories, bus, and the main memory
- Small benchmark programs that can fit in cache memory may execute from a cache memory with negligible use of bus and main memory

# Typical floating point benchmarks

- Matrix inversion and other matrix operations
- Solution of linear algebraic equations
- Roots of polynomials
- Numeric integration
- Computation of functions

# Matrix inversion benchmark

- Rule #1: Avoid using matrices that contain random numbers – such matrices are frequently singular or can cause other numerical problems
- Use matrices that are proved to be regular.
- Strictly diagonally dominant matrices are non-singular

# Strictly diagonally dominant matrix

- Strictly diagonally dominant matrix satisfies the condition

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}|, \quad i = 1, \ldots, n, \quad j = 1, \ldots, n$$

- This matrix is nonsingular (invertible), and therefore it is convenient for benchmarking
- There are other matrices that are also known as nonsingular

# Invertible matrices that can cause numeric problems in benchmarking

- Cauchy's matrix

```
a[i][j] = 1/(i+j)
```

- Hilbert's matrix

```
a[i][j] = 1/(i+j-1)
```

- Vandermonde's matrix

```
a[i][j] = j**i
```

# An invertible combinatorial matrix that is suitable for benchmarking

```
a[i][j] = 1.0001, i≠j
        = 2.0001, i=j  (any size)
```

$$
\begin{bmatrix}
2.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 \\
1.0001 & 2.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 \\
1.0001 & 1.0001 & 2.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 \\
1.0001 & 1.0001 & 1.0001 & 2.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 \\
1.0001 & 1.0001 & 1.0001 & 1.0001 & 2.0001 & 1.0001 & 1.0001 & 1.0001 \\
1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 2.0001 & 1.0001 & 1.0001 \\
1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 2.0001 & 1.0001 \\
1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 1.0001 & 2.0001
\end{bmatrix}
$$

Sample matrix of size n=8

# Single index matrices

- In benchmarking it is frequently necessary to adjust the size of matrix to achieve a desired run time or other properties
- Matrices with 2 indices are not suitable for frequent changes of size
- Single-index matrices are just a single array and their size is easily adjusted. They are convenient for benchmarking.

# Avoid using library functions

- Library functions from math.h, such as sin, cos, exp, log, etc. differ from compiler to compiler, and therefore may be a source of differences in seemingly equal workloads
- Another function that is also machine-dependent is rand( ) from stdlib.h
- Library functions should be used in benchmarking only when natural workloads use library functions with high frequency

# Typical integer/combinatorial benchmarks

- Sorting arrays
- Search algorithms
- Random number generation and testing
- Combinatorial optimization
- Puzzles (e.g. 8 queens)
- Recursive algorithms

# What data are convenient for sorting in benchmark programs?

- Random numbers can be convenient for sorting provided that they are fully reproducible on all machines starting with those having 16 bit integers.
- Deterministic arrays can also be suitable provided that they contain some regular nontrivial pattern

# Random number generators for benchmarking

- Random number generators for benchmarking must satisfy the following conditions:
  - Same behavior on machines having the word size of 16 bits and more
  - Long cycle of random numbers
  - Simplicity and speed
  - Quality of distribution of random numbers is not important

# Fibonacci-style generators

- $N(i) = (N(i-1) + N(i-2)) \bmod M, \quad i=2,3,\ldots$
- Very simple and fast
- We must select N(0), N(1), and M
- Cycle repeats when

  $N(c)=N(1)$ and $N(c-1)=N(0)$

  The cycle size is c.
- If working with 16-bit integers the condition for avoiding overflow is

  $N(i-1) + N(i-2) < 2^{14} = 16384$

# What are the best values for N(0), N(1), and M?

- Theoretical answer to this question is not simple [Knuth, Vol. 2]
- Empirical answer to this question is rather simple and based on experiments with a program that computes the maximum cycle length for various values of N(0) and N(1)
- In the following program we use p=N(i-2), q=N(i-1), r=N(i)

```
#include<iostream.h>
#include<stdlib.h>
#include<iomanip.h>
void main(void)  // Finding maximum cycle of a Fibonacci-style random number generator
{          unsigned long int cycle=0, cycle_max=0;
           unsigned int M_init=10000, M, N0=1, N1=2, M_opt, p, q, r;
           for(int k=0; k<20; k++)
           {          for(M=M_init; M<16384; M++)  // 2^14=16384 or 2^15=32768
                      {          p=N0; q=N1; cycle=0;
                                 do
                                 {          r = (p+q) % M; cycle++;
                                            p = q;  q = r;
                                 }while (!((p == N0) && (q == N1)));
                                 if(cycle>cycle_max) {cycle_max = cycle; M_opt = M;}
                      }
                      cout << "N0=" << setw(5) << N0 <<"   N1=" << setw(5) << N1
                           <<"   M_opt="<< setw(6) <<M_opt
                           <<"   Max_cycle="<< setw(6) << cycle_max <<"\n";
                      N0=rand()%M_init; N1=rand()%M_init;
           }
}
```

## Optimum modulus and maximum cycle size for 20 random seeds in the case of integers where M<16384

```
N0=    1   N1=    2   M_opt= 15625   Max_cycle= 62500
N0=   41   N1= 8467   M_opt= 15625   Max_cycle= 62500
N0= 6334   N1= 6500   M_opt= 15625   Max_cycle= 62500
N0= 9169   N1= 5724   M_opt= 15625   Max_cycle= 62500
N0= 1478   N1= 9358   M_opt= 15625   Max_cycle= 62500
N0= 6962   N1= 4464   M_opt= 15625   Max_cycle= 62500
N0= 5705   N1= 8145   M_opt= 15625   Max_cycle= 62500
N0= 3281   N1= 6827   M_opt= 15625   Max_cycle= 62500
N0= 9961   N1=  491   M_opt= 15625   Max_cycle= 62500
N0= 2995   N1= 1942   M_opt= 15625   Max_cycle= 62500
N0= 4827   N1= 5436   M_opt= 15625   Max_cycle= 62500
N0= 2391   N1= 4604   M_opt= 15625   Max_cycle= 62500
N0= 3902   N1=  153   M_opt= 15625   Max_cycle= 62500
N0=  292   N1= 2382   M_opt= 15625   Max_cycle= 62500
N0= 7421   N1= 8716   M_opt= 15625   Max_cycle= 62500
N0= 9718   N1= 9895   M_opt= 15625   Max_cycle= 62500
N0= 5447   N1= 1726   M_opt= 15625   Max_cycle= 62500
N0= 4771   N1= 1538   M_opt= 15625   Max_cycle= 62500
N0= 1869   N1= 9912   M_opt= 15625   Max_cycle= 62500
N0= 5667   N1= 6299   M_opt= 15625   Max_cycle= 62500
```

```
N0=    1   N1=    2   M_opt= 31250   Max_cycle=187500
N0=   41   N1= 8467   M_opt= 31250   Max_cycle=187500
N0= 6334   N1= 6500   M_opt= 31250   Max_cycle=187500
N0= 9169   N1= 5724   M_opt= 31250   Max_cycle=187500
N0= 1478   N1= 9358   M_opt= 31250   Max_cycle=187500
N0= 6962   N1= 4464   M_opt= 31250   Max_cycle=187500
N0= 5705   N1= 8145   M_opt= 31250   Max_cycle=187500
N0= 3281   N1= 6827   M_opt= 31250   Max_cycle=187500
N0= 9961   N1=  491   M_opt= 31250   Max_cycle=187500
N0= 2995   N1= 1942   M_opt= 31250   Max_cycle=187500
N0= 4827   N1= 5436   M_opt= 31250   Max_cycle=187500
N0= 2391   N1= 4604   M_opt= 31250   Max_cycle=187500
N0= 3902   N1=  153   M_opt= 31250   Max_cycle=187500
N0=  292   N1= 2382   M_opt= 31250   Max_cycle=187500
N0= 7421   N1= 8716   M_opt= 31250   Max_cycle=187500
N0= 9718   N1= 9895   M_opt= 31250   Max_cycle=187500
N0= 5447   N1= 1726   M_opt= 31250   Max_cycle=187500
N0= 4771   N1= 1538   M_opt= 31250   Max_cycle=187500
N0= 1869   N1= 9912   M_opt= 31250   Max_cycle=187500
N0= 5667   N1= 6299   M_opt= 31250   Max_cycle=187500
```

Optimum modulus and maximum cycle size for 20 random seeds in the case of unsigned integers where M<32768

# Machine-independent random number generator working with integers

```
int rng_i( )                    // Random integers
{                               // Cycle size = 62500
    static int p=1, q=2, r;
    r = (p+q) % 15625; p=q; q=r;
    return r;
}
```

# Machine-independent random number generator working with unsigned integers

```
unsigned int rng_ui( )  // Unsigned random integers
{                       // Cycle size = 187500
    static unsigned int p=1, q=2, r;
    r = (p+q) % 31250; p=q; q=r;
    return r;
}
```

# Properties of rng_i( ) and rng_ui( )

- Portable generator
- Reproducible Fibonacci-style random sequence
- Sequence length = 62500 or 187500
- No overflow even with 16-bit machines
- Suitable for sort benchmarks

# Testing rng_ui and rng_i

- Generate 62500 random numbers with rng_i
- Generate 187500 random numbers with rng_i
- In both cases the last two numbers must be 1 and 2

```
int rng_i( )             // Random integers: cycle size = 62500
{        static int p=1, q=2, r;
         r = (p+q)%15625; p=q; q=r;
         return r;
}
unsigned int rng_ui( )  // Unsigned random integers: cycle size = 187500
{        static unsigned int p=1, q=2, r;
         r = (p+q)%31250; p=q; q=r;
         return r;
}
void main(void) // Testing rng_i and rng_ui
{        for(int i=0; i<62498; i++) rng_i();
         cout << rng_i(); cout << ' ' << rng_i() << endl;

          for(i=0; i<187498; i++) rng_ui();
         cout << rng_ui(); cout << ' ' << rng_ui() << endl;
}
1 2
1 2
```

# Deterministic sequences

- Deterministic sequences can be generated by repeating the integer expression n = n+1 – n/M*M. The resulting sequence is:

  1,2,…,M, 1,2,…,M, …

- Sample generator loop:

  for(k=0; k<kmax; k++) a[k]=n=n+1-n/M*M;

- Such sequences can be sorted in decreasing order

# Some simple algorithms that can be used as small CPU benchmarks

# Linear Algorithm: Factorial

```
int f(int n)
{    return (n<1) ? 1 : (n*f(n-1));
}
int F(int n)
{    int f=1, i;
     for(i=2; i<=n; i++) f *= i;
     return f ;
}
```

Useful for very short run times

$T(n) = O(n)$ (both iterative and recursive)

# Linear Algorithm: Two Variables

```
void merge (int a[ ], int na, int b[ ], int nb, int c[ ], int& nc)
{ int  i,j ;
   nc=i=j=0 ;
  while (i<na && j<nb) c[nc++] = (a[i]<b[j]) ? a[i++] : b[j++];
  while (i<na)         c[nc++] = a[i++] ;
  while (j<nb)         c[nc++] = b[j++] ;
}
```

Useful for very short run times

Run time: $T(na, nb) = c \bullet nc = c(na+nb)$

# Logarithmic Algorithm: Binary Search

```
int bsearch(int v[], int n, int x)
{ int low, high, mid ;
  low=0 ; high = n-1 ;
  while (low <= high)
  {  mid = (low + high) / 2 ;
     if       (x < v[mid])  high = mid-1 ;
     else if (x > v[mid])  low  = mid+1 ;
     else return mid ;
  }
  return -1 ;  /* no match */
}
```

Useful for very short run times: O(log n). One of the most popular algorithms in Computer Science

# Recursive Binary Search

```
int bsearch(int v[ ], int low, int high, int x)
{int mid = (low + high) / 2;
 if(low>high) return -1;
 if(x<v[mid]) return bsearch(v, low, mid-1, x);
 if(x>v[mid]) return bsearch(v, mid+1, high, x);
 return mid;
}
```

Useful for testing recursion.
Very short run times: O(log n).

# Quicksort: O( n $\log_2(n)$)

```
void sort(int a[], int left, int right)
{ int i, mid;

  if(left >= right) return;

  for(mid=left, i=left+1; i <= right; i++)

    if(a[i] < a[left]) swap(a, ++mid, i);

  Swap(v, left, mid);

  sort(v,left, mid-1);

  sort(a, mid+1, right);

}
```

One of the most popular algorithms in Computer Science. Frequently used and useful for benchmarking

# Quadratic Algorithms: $O(n^2)$

For each component of data set process all components of the data set:

```
for(i=0; i<n; i++)
   for(i=0; i<n; i++)
      {Processing}
```

These algorithms may be useful for easy increase of run time by increasing n.

$$T(n) = c \bullet n \bullet n = O(n^2)\ ; \quad c = \text{const.}$$

# Simple Select Sort

```
void sort( int a[], int n )
{ int i, j;
    for( i=0 ; i<n-1 ; i++ )
       for( j=i+1 ; j<n ; j++ )
           if( a[i] > a[j] ) swap(a[i], a[j]);
}
```

Inefficient but extremely simple algorithm. Sometimes used for simplistic benchmarking.

$T(n) \approx c(\text{number of executed if statements}) = c(1 + 2 + \ldots + n\text{-}1)$

$= cn(n-1)/2 = c(n^2-n)/2$

$T(n) = O(n^2)$

# Quadratic algorithm: Matrix Initialization

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        a[i][j] = rand( );
```

$T(n) = c\ n^2 = O(n^2\ )$

# Quadratic algorithm: Matrix Transposition

```
for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
        swap(a[i][j] , a[j][i]);
```

$$T(n) = c(1+2+\ldots+n-1) = c\,(n^2 - n) / 2 = O(n^2)$$

# Matrix Multiplication

The most frequent matrix operation. Easy to increase the run time by adusting n.

```
for(i=0; i<n; i++)
     for(j=0; j<n; j++)
          for(c[i][j] = k = 0; k<n; k++)
               c[i][j] += a[i][k]*b[k][j];
```

$$T(n) = c \bullet n \bullet n \bullet n = O(n^3)\ ; \quad c = \text{const.}$$

# Exponential Algorithm: Fibonacci Numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
int f(int n)
{
    return (n<2) ? n : f(n-1)+f(n-2) ;
}
```

Extremely inefficient and almost illegal in programming practice. Can be used for testing recursion only.

$$
\begin{aligned}
T(n) &= t + T(n-1) + T(n-2) \approx T(n-1) + T(n-2)\,, \quad n > 1 \\
&= 1\,, \quad n = 1 \\
&= 0\,, \quad n = 0
\end{aligned}
$$

# Solving T(n) = T(n-1) + T(n-2)

Suppose that $T(n) = cg^n, \quad g > 0$

$$cg^n = cg^{n-1} + cg^{n-2}$$

$$1 = g^{-1} + g^{-2}$$

$$g^2 - g - 1 = 0$$

$$g = \frac{1 \pm \sqrt{1+4}}{2}; \quad g = \frac{1+\sqrt{5}}{2} = 1.618$$

$$T(n) = c\,1.618^n = O(1.618^n)$$

# Linear Fibonacci Numbers

```
int f(int n)
{ int i, zero=0, first=1, second=n;
  for(i=2; i<=n); i++)
  {   second = first + zero;
      zero = first;  first = second;
  }
  return second;
}
```

This is a proper way to compute Fibonacci numbers. Insufficient complexity and too short run time make this program unsuitable for benchmarking.

$$T(n) = c \bullet n = O(n)\ ; \quad c = \text{const.}$$

# How to make a SpeedMark benchmark

```
// Pseudocode of Speedmark benchmark (HW#1)
// 1. Run 10 seconds matrix inversion (or similar workload)
// 2. Compute matrix inversion (floating point) speed
// 3. Run 10 seconds quicksort (or similar workload)
// 4. Compute quicksort (integer) speed
// 5. Compute and display the average speed in operations/min


   int NINT=0, NFLOAT=0;
   double START, VINT, VFLOAT, AverageSpeed;

   START = sec( );
   while(sec( ) < START+10) {Minv( ); NFLOAT++;}
   VFLOAT = 60*NFLOAT/(sec( )-START);

   START = sec( );
   while(sec( ) < START+10) {Qsort( ); NINT++;}
   VINT = 60*NINT/(sec( )-START);

   AverageSpeed = 2*VFLOAT*VINT/(VFLOAT+VINT);
   Display VFLOAT, VINT, AverageSpeed;
// Select Minv( ) and Qsort( ) so that they run between 1 and 3 sec
```

# Average speed as a harmonic mean

$VFLOAT =$ speed of floating point operations

$VINT =$ speed of integer operations

$$AVERAGE\_SPEED = \frac{2}{\dfrac{1}{VFLOAT} + \dfrac{1}{VINT}}$$

$$= \frac{2 \times VFLOAT \times VINT}{VFLOAT + VINT}$$

# Disk benchmarks

# Default workloads

- Writing a sequential file
- Reading a sequential file
- Random writing in a relative file
- Random reading from a relative file

# Problems

- What is the available space on disk?
- How big should be the file?
- Contiguous or fragmented files?
- How to compare disks that have different capacities?

# Disk speed as a harmonic mean

$VSEQ =$ speed of sequential read operations

$VRAN =$ speed of random read operations

$$AVERAGE\_SPEED = \frac{2}{\dfrac{1}{VSEQ} + \dfrac{1}{VRAN}}$$

$$= \frac{2 \times VSEQ \times VRAN}{VSEQ + VRAN}$$

# Other disk operations

- Disk sequential write (SEQW)
- Disk sequential read (SEQR)
- Disk random write (RANW)
- Disk random read (RANR)

# Disk speed as a harmonic mean
# (a) The case of equal weights

$VSEQW =$ speed of sequential write operations

$VRANW =$ speed of random write operations

$VSEQR =$ speed of sequential read operations

$VRANR =$ speed of random read operations

$$AVERAGE\_SPEED = \frac{4}{\frac{1}{VSEQR} + \frac{1}{VRANR} + \frac{1}{VSEQW} + \frac{1}{VRANW}}$$

# Disk speed as a harmonic mean (b) The case of different weights

$VSEQW =$ speed of sequential write operations

$VRANW =$ speed of random write operations

$VSEQR =$ speed of sequential read operations

$VRANR =$ speed of random read operations

$$AVERAGE\_SPEED = \frac{1}{\dfrac{WSEQR}{VSEQR} + \dfrac{WRANR}{VRANR} + \dfrac{WSEQW}{VSEQW} + \dfrac{WRANW}{VRANW}}$$

$$WSEQR > 0, \quad WRANR > 0, \quad WSEQW > 0, \quad WRANW > 0$$

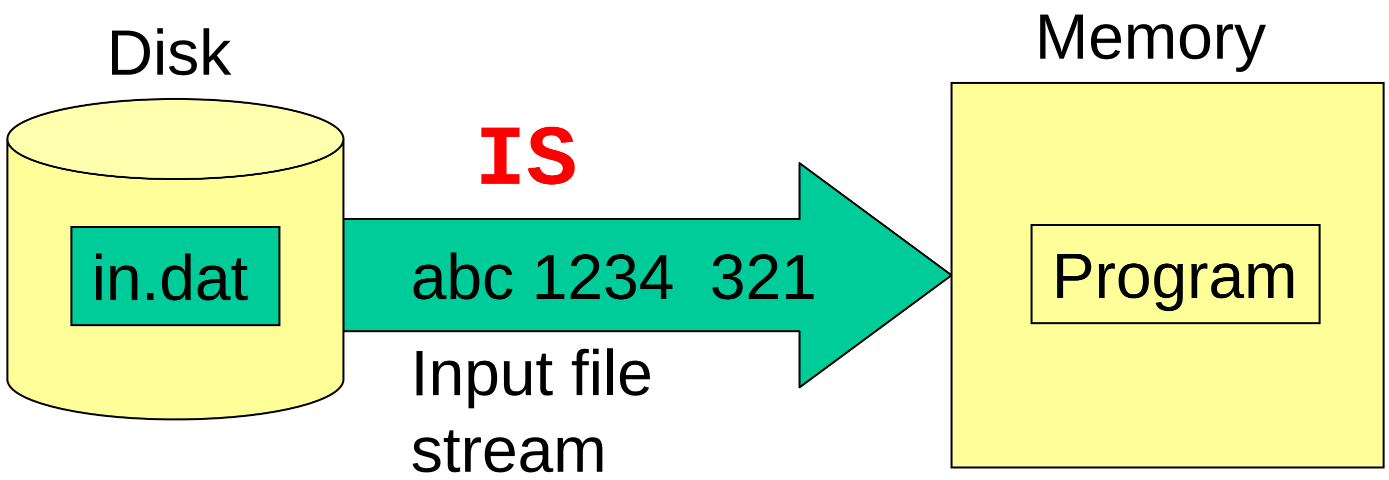
$$WSEQR + WRANR + WSEQW + WRANW = 1$$

# Using different weights

- Weights determine relative importance
- Generally, the frequency of sequential and random disk operations is not the same
- Generally, the frequency of read and write disk operations is not the same
- Different weights can be used to take into account the different frequency of use

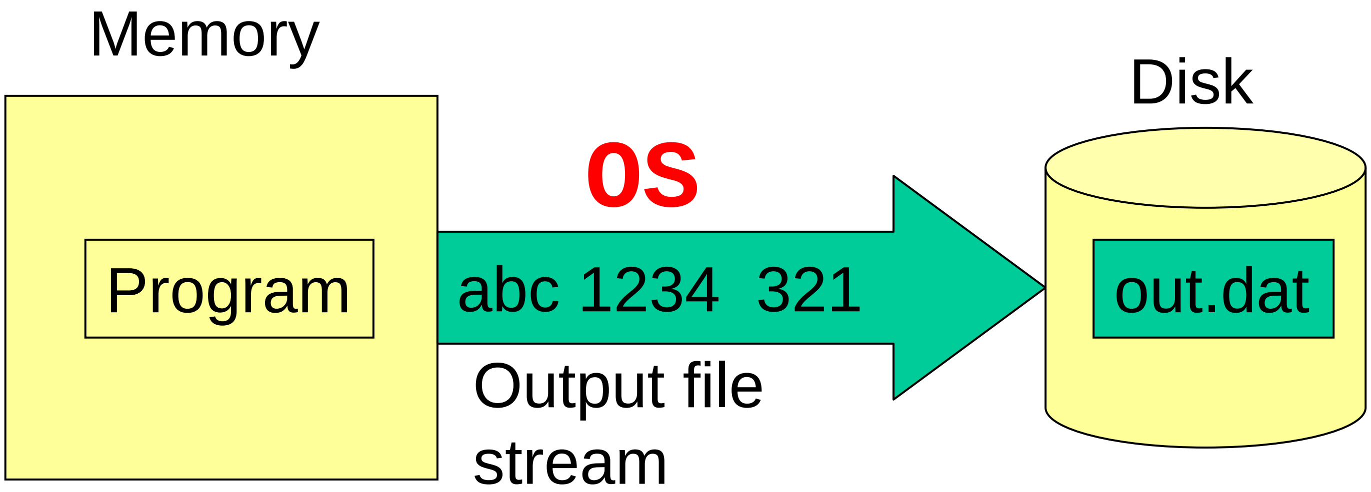
# Review of basic disk operations in C++

# Input File Stream



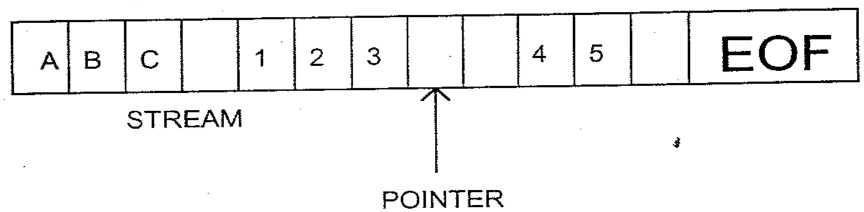

Input file stream is a sequence of bytes (characters) generated when we read data from a disk file. In C++ the input file stream can have any name: e.g., **IS**.

# Output File Stream




Output file stream is a sequence of bytes (characters) generated when we write data to a disk file. In C++ the output file stream can have any name: e.g., **OS**.

# FILES




Stream = sequence of bytes and a pointer showing the current byte (where insert/extract can start)

FILE = stream of characters terminated by the End Of File (EOF) record; also called ***file stream***

Location: Disk, Floppy disk, tape, CD ROM, (sometimes even in memory), etc.

Size: Limited only by medium capacity

# C++ FILE OPERATIONS

Library: #include <fstream.h> // Support for file stream operations

Declaration: ifstream IS; // Input file stream
ofstream OS; // Output file stream

File name: (1) internal, and (2) external

Internal name: IS, and OS are **internal names** used inside a C++ program to refer to file streams. Same rules as for any variable name.

External name: depends on rules for file names used by the operating system

**char EFname[ ] = "data.txt";**

OPEN FILE: includes several operations:
- connect internal name IS and external name "data.txt"
- check access right of the user
- create buffer areas in memory
- transfer directory record from directory file to operating system

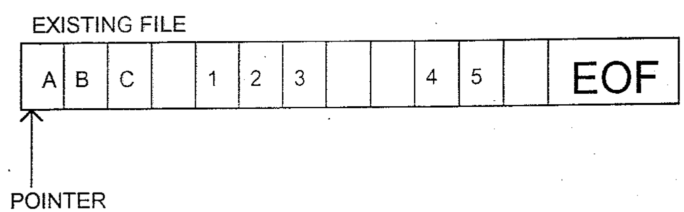
CLOSE FILE: includes several operations:
- disconnect internal name IS and external name "data.txt" (IS becomes an undefined variable)
- release buffer areas in memory
- transfer updated directory record from the OS to the directory file

General rule: OPEN as late as possible, and CLOSE as soon as possible, but do not exagerate! Power failure or OS crash when a file is open can cause some data to be lost.

Purpose: File can be open for three purposes defined using input/output stream (ios) class member functions:

- read **ios::in**
- write **ios;:out**
- append **ios::app**

**READ:**




**IS.open(EFname, ios::in);** // open for read

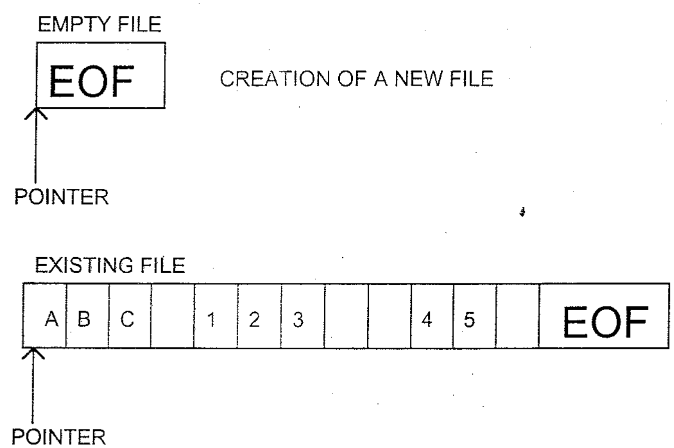
Declaration with initialization:

for Borland compilers

**ifstream IS = EFname ;** // shortest form, but NOT STANDARD

**ifstream IS(EFname, ios::in) ;** ← STANDARD

## WRITE:



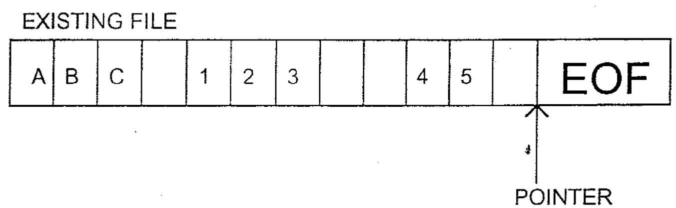

**OS.open(EFname, ios::out);** // open for write

If the file **EFname** exists prior to execution of the **open( )** function, then its contents is lost because EOF will be moved at the beginning of the file (**ios::out** is destructive!)

Declaration with initialization:

**ofstream OS = EFname ;** (NOT STANDARD!)
**ofstream OS(EFname, ios::out) ;**

## APPEND:




**OS.open(EFname, ios::app);** // open for append

**EFname** exists prior to execution of the **open( )** function, and its contents is preserved

Declaration with initialization:

**ofstream OS(EFname, ios::app) ;**

## EXTRACT (READ) DATA

```
int data;
char c, s[80];

IS >> data ;

IS.getline(s, 80);
IS.get(c);
```

**(1) Skip white spaces (blanks, tabs, newlines, formfeeds, etc.)**

**(2) Extract (and convert) data as long as possible**

**(3) Stop conversion when a white space or non-convertible character is encountered (this also includes the EOF record)**

CLOSE FILE:

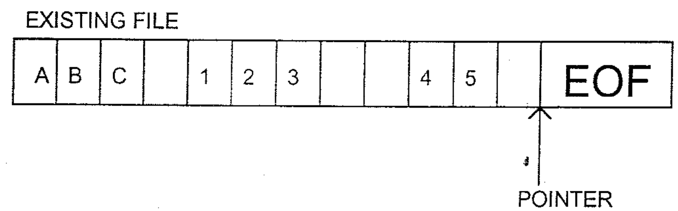
```
IS.close( ) ;  //  IS becomes undefined

OS.close( );  //  OS becomes undefined
```

DETECT EOF:

```
IS >> data ;
while(! IS.eof( ))
{
    // Process data
    IS >> data ;
}
```

Function **eof( )** returns true (1) if EOF is deteced

# INSERT (WRITE) DATA




**OS << data ;**

**OS << s ;** // write a line into file
**OS.put(c) ;** // write one character

## TEST OPEN FILE SUCCESS/FAILURE

Reason for failure:

- no file
- wrong directory
- no medium
- hardware error

TEST FUNCTION:

**IS.fail( )**
**OS.fail( )**

**0** = successful last operation with IS/OS
**!0 (usually 1)** = last operation with IS/OS failed

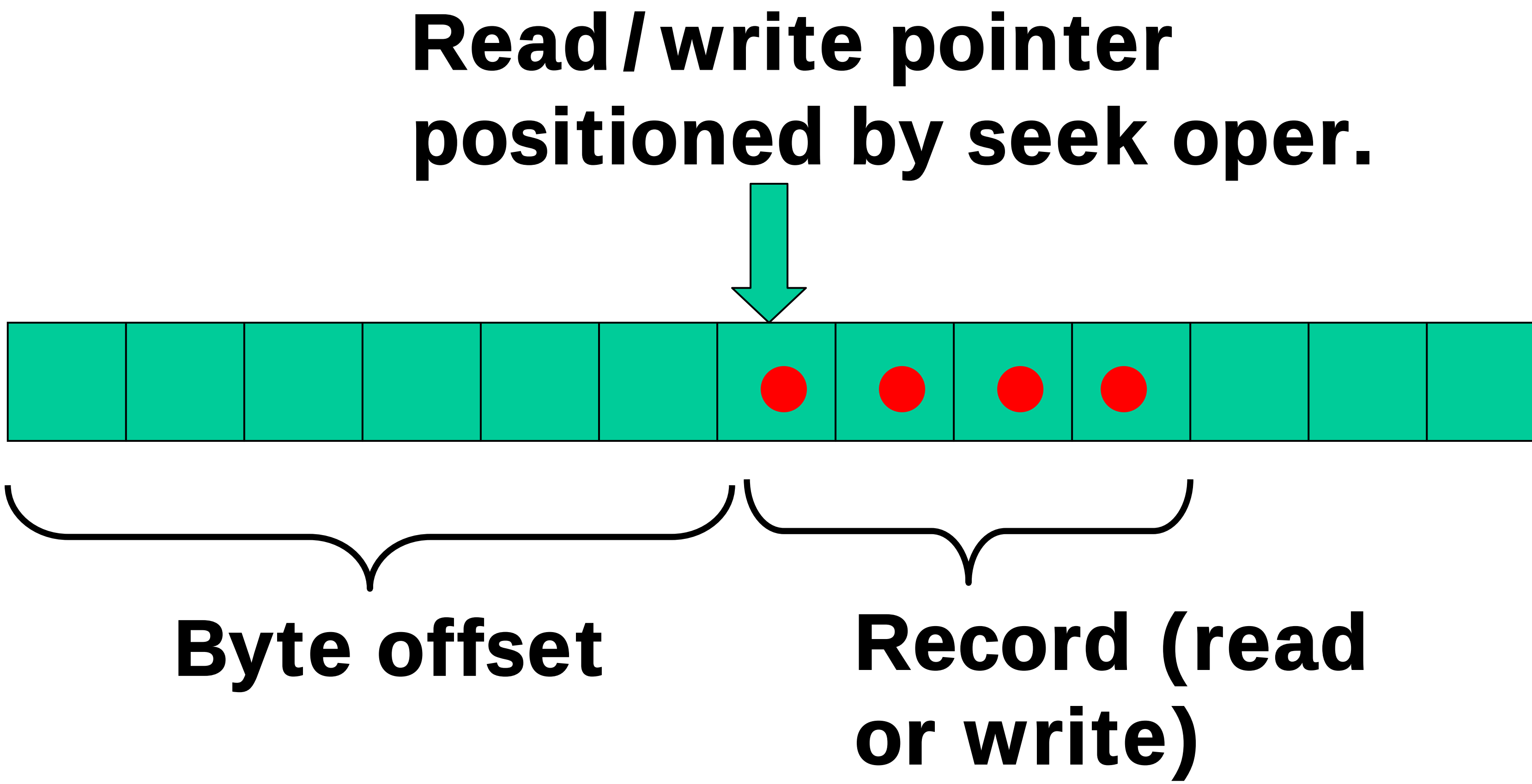
**PASSING OF FILE STREAM ARGUMENTS**
**(as other compound objects - only by reference)**

```
void  io(ifstream& IS, ofstream& OS, double& data)
{
   int a, b, c;
   IS >> a >> b >> c ;  // READ FROM IS
   OS << a << b ;       // COPY TO OS
   data = a+b+c;        // RETURN VALUE
}                    // IS and OS are formal streams
                     (they do not exist as data objects)
void main(void)
{
   int result ;
   ifstream INPUT = "in.dat" ;   // actual input stream
   ofstream OUTPUT = "out.dat" ; // actual out stream
   io(INPUT, OUTPUT, result) ;  // Function call
   ..............
   INPUT.close( ) ;        // if close is omitted, it will
   OUTPUT.close( ) ;       // be done automatically at
   ..............          // the end of main program
}
```

# Relative Files

- Relative files consist of indexed records
- Relative files support direct access (access according to the byte offset: the position is defined as the distance in bytes from the beginning of file)
- Each record can be designed as a struct

# File model: a sequence of bytes



# Record structure and size

- Example:

```
struct client
{ int   account;
  char name[20];
  double balance;
} ;
```

- Size of record is measured in bytes and determined using the sizeof( ) function: sizeof(client)

# Initialization of record

```
client  c ;
client blank = { 0, “”, 0.}

cin >> c.account
    >> c.name
    >> c.balance ;
```

# Basic operations: write

- Opening a relative file for writing:
  ofstream OS(filename, ios::ate)
  (ate = positioning “at end”)
- Writing records in arbitrary order:
  – Selecting file record (positioning file pointer)
    OS.seekp(<byte offset>)
  – OS.write(<record pointer>, <record size>)
- seekp : positioning for put operation

# Basic operations: read

- Opening a relative file for reading:
  ofstream OS(filename, ios::in) (seq)
  ofstream OS(filename, ios::ate) (random)
- Reading records in arbitrary order:
  - Selecting file record (positioning file pointer)
    OS.seekg(<byte offset>)
  - OS.read(<record pointer>, <record size>)
- seekg : positioning for get operation

# Cast operator (type)

- Conversion to desired data type
- Syntax:

  (desired data type) expression

- Cast is a unary operator and has higher precedence than arithmetic operations
- Example:

  X = (double) (n + 1)

- Alternative way:

  X = double (n + 1)

# More casting

- int k=1, n=2;
  double x = (double) k/n  // x = 0.5
  // The first operation is the higher
  // precedence unary (double)k
  // Then n is automaticaly promoted to
  // double
- (char *) &<struct record> // address of an arbitrary record converted to a character pointer

# Record pointer

- The type of record pointer is char *
- The address  &<struct record> has the type different from char *
- Conversion to type char * is performed using cast (char *) &<struct record>
-  IS.read((char *) &<rec>, sizeof(rec))
- OS.write((char *) &<rec>, sizeof(rec))