CHAPTER 2
ARRAYS

Iris Hui-Ru Jiang

Arrays

- **Contents**
  - ADT vs. C++ class
  - Array as ADT
  - Array applications
    - Ordered lists, e.g., polynomials
    - Sparse matrices
  - Array representation
  - Strings
    - String pattern matching
- **Readings**
  - Chapter 2
  - C++ STL
    - Array: class vector
    - String: classes basic_string<T>, string, wstring

C++ Class (1/2)

Define in *Rectangle.h*

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include <iostream.h>
class Rectangle {
friend ostream& operator<< (ostream& os,
      Rectangle& r);
public:       // public members
    // member functions
    Rectangle(int x=0, int y=0, int h=0, int w=0);
                // constructor w/ or w/o arguments
    ~Rectangle();           // destructor
    int GetHeight();        // return height
    int GetWidth();         // return width
private:      // private members
    // data members
    int xLow, yLow, height, width;
    // (xLow, yLow) -- bottom left corner
};
#endif
```

specification

implementation

Arrays

Components of a class

- **Class name**
  - *Rectangle*
- **Data members**
  - *xLow*, *yLow*, *height*, *width*
- **Member functions**
  - *Rectangle*(), ~*Rectangle*(), *GetHeight*(), *GetWidth*()
- **Levels of program access**
  - **public**: anywhere
  - **protected**: within it or from subclasses or by a **friend**
  - **private**: within it or by a **friend**

Data encapsulation in C++:
Declare all data members as private/protected

C++ Class (2/2)

Implement in *Rectangle.cpp*

```
#include "Rectangle.h"
// "Rectangle::" identifies member functions
Rectangle::Rectangle(
    int x=0, int y=0, int h=0, int w=0)
: xLow(x), yLow(y), height(h), width(w)
{ }
```

constructor

```
Rectangle::~Rectangle()
{ }

int Rectangle::GetHeight() { return height;}
int Rectangle::GetWidth() { return width;}

ostream& operator<<(ostream& os, Rectangle& r)
{
    os << "Position is: " << r.xLow << " ";
    os << r.yLow << endl;
    os << "Height is: " << r.height << endl;
    os << "Width is: " << r.width << endl;
    return os;
}
```

1. overloading
2. friend

Declare & Invoke in *main.cpp*

```
#include <iostream.h>
#include "Rectangle.h"

main () {
    Rectangle r (1, 3, 6, 6); // r: class object
    Rectangle s;              // s: class object
    Rectangle *t = &s;        // t: pointer to s

    // use . to access members of class objects
    // use -> to access them through pointers
    if (r.GetHeight()*r.GetWidth() >
        t->GetHeight()*t->GetWidth())
        cout << "r ";
    else cout << "s ";
    cout << "has the greater area" << endl;
    cout << "r:" << endl;
    cout << r;
}
```

Separate member function
Implementation from class definition

ADT *NaturalNumber*

ADT

ADT *NaturalNumber* is
objects: An ordered subrange of the integers starting at zero and ending at MAXINT.
functions: for all *x*, *y* ∊ *NaturalNumber*; TRUE, FALSE ∊ *Boolean*, +, -, <, ==, and = are the usual integer operations

```
    Zero(): NaturalNumber ::= 0
    IsZero(x): Boolean ::=
    if (x==0) IsZero=TRUE else IsZero=FALSE
    Add(x, y): NaturalNumber ::=
    if (x+y<=MAXINT) Add=x+y     else Add=MAXINT
    Equal(x, y): Boolean ::=
    if (x==y) Equal=TRUE else Equal=FALSE
    Successor(x): NaturalNumber ::=
    if (x==MAXINT) Successor=x else Successor=x+1
    Substract(x, y): NaturalNumber ::=
    if (x<y) Substract = 0 else Substract=x–y
end NaturalNumber
```

C++ Class

```
class NaturalNumber {
// An ordered subrange of the integers starting at
      zero and ending at MAXINT.
public:

    NaturalNumber Zero(); // return 0
    bool IsZero();
    // if *this is 0, return true; or, false
    NaturalNumber Add(NaturalNumber y);
    // return min(*this+y, MAXINT)
    bool Equal(NaturalNumber y);
    // if *this==y, return true; or, false
    NaturalNumber Successor();
    // if *this is MAXINT, return MAXINT; or, *this+1
    NaturalNumber Subtract(NaturalNumber y);
    // if *this < y, return 0; or, *this–y
};
```

***this** points to "this" object

Use C++ class to define an ADT instead

Array as ADT

Considering Array as ADT

class *GeneralArray* **{**
// A set of pairs <*index*, *value*> where for each value of *index* in *IndexSet*,
// there is a *value* of type **float**. *IndexSet* is a finite ordered set of one or more dimensions,
// for example, {0, …, *n*-1} for one dimension,
// {(0, 0), (0, 1), (0, 2),(1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.
public:
    GeneralArray(**int** *j*, *RangeList list*, **float** *initValue* = defatultValue)**;**
    // This constructor creates a *j* dimensional array of **float**s;
    // the range of the *k*th dimension is given by the *k*th element of *list*.
    // For each index *i* in the index set, insert <*i*, *initValue*> into the array.

    float *Retrieve*(*index i*)**;**
    // if (*i* is in the index set of the array) return the float associated with *i* in the array;
    // otherwise throw an exception

    void *Store*(*index i*, **float** *x*)**;**
    // if *i* is in the index set of the array, replace the old value associated with *i* by *x*;
    // otherwise throw an exception.
}; // end of *GeneralArray*

Array is more than “a consecutive set of memory locations”
Array can be viewed as a mapping, a set of pairs <index, value>

Why Not Just Using C++ Array?

- ***GeneralArray* is more general than a C++ array**
  - C++ array
    - Declaration: **float** *floatArray*[*n*]**;**
    - Access *i*th element: *floatArray*[*i*] or *(*floatArray* + *i*)
  - C++ array requires the index set to be a set of consecutive integers starting at 0
  - C++ does not check an array index to ensure that it belongs to the range for which the array is defined

Array Applications: Ordered Lists

Array Is Not Just “Array”

- **Applications:** ordered (linear) lists, etc.
- **Examples:**
  - Days of the week: (Sun, Mon, Tue, Wed, Thu, Fri, Sat)
  - Values of poker cards: (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K)
  - Seasons: (spring, summer, fall, winter)
  - Floors of a building: (basement, lobby, mezzanine, first, second)
  - Years the US fought in WWII: (1941, 1942, 1943, 1944, 1945)
  - Years Switzerland fought in WWII: () -- empty list!
  - Polynomials: $a(x) = 3x^2 + 2x - 4$

Operations on Lists

- **Operations**
  1. Find the length, n, of the list
  2. Read the list from left to right (or right to left)
  3. Retrieve the i^{th} element, $0 \le i < n$
  4. Store a new value into the i^{th} position, $0 \le i < n$
  5. Insert a new element at the i^{th} position, $0 \le i < n$
  6. Delete the i^{th} element, $0 \le i < n$
- **How to implement it efficiently?**
  - Array? Allocate consecutive memory locations
    - Associate the list element a_i with the array index i
    - Retrive/Store: $O(1)$
    - Insert/Delete: $O(n)$
  - Other options detailed in Chapter 4

Array Applications: Polynomials

Polynomials

- **Example:**
  - $a(x) = \sum a_i x^i = 3x^2 + 2x - 4$
  - $b(x) = \sum b_i x^i = x^8 - 10x^5 - 3x^3 + 1$
- **A term:** (coeficient, exponent)
  - Nonzero term: a term has a nonzero coeficient
  - e.g., $3x^2$: (3, 2)
- **Degree:** largest exponent among nonzero terms
  - e.g., degree of $a(x) = 2$, degree of $b(x) = 8$
- **Sum:** $a(x)+b(x) = \sum (a_i+b_i)x^i$
- **Product:** $a(x)*b(x) = \sum (a_i x^i * (\sum b_j x^j))$

ADT *Polynomail*

class *Polynomial* **{**
// $p(x) = a_0x^{e0}+\ldots+a_nx^{en}$; a set of ordered pairs of $<e_i, a_i>$,
// where a_i is a nonzero float coefficient and e_i is a non-negative integer exponent.
public:

Polynomial();
// Construct the polynomial $p(x) = 0$

Polynomial Add(*Polynomial poly*)**;**
// Return the sum of the polynomials ***this** and *poly*

Polynomial Mult(*Polynomial poly*)**;**
// return the product of the polynomials ***this** and *poly*

float *Eval*(**float** *f*);
// Evaluate the polynomial ***this** at *f* and return the result

}; // end of *Polynomial*

Polynomial Representations (1/2)

Representation 1

private:
    int *degree*; // *degree* ≤ *MaxDegree*
    float *coef* [*MaxDegree* + 1]; // coefficient array

- *MaxDegree*: constant: allowed largest degree
- *Polynomial* class object *a* of degree *n* ≤ *MaxDegree*:
  a.*degree* = *n*
  a.*coef*[*i*] = a_{n-i}, $0 \le i \le n$ // in descending order
- Simple, but…(size of *coef*)
  - We have to know *MaxDegree*
  - May waste memory usage
    - e.g., *MaxDegree* = 1,000,000 and *n* = 2, say $a(x)=2x^2+1$

Representation 2

private:
    int *degree*;
    float **coef*;

constructor

Polynomial::*Polynomial*(**int** *d*)
{
    degree = *d*;
    coef = **new float** [*degree*+*1*];
}

- Size of *coef* = *a*.*degree*+1
  - Dynamic allocation
- What if a sparse polynomial?
  - e.g., $a(x)=2x^{1000}+1$, with many zero terms

Polynomial Representations (2/2)

Representation 3

```
class Polynomial;  // forward delcaration

class Term {
friend Polynomial;
private:
    float coef; // coefficient
    int exp; // exponent
};

// The private data members of Polynomial
private:
    Term *termArray; // array of nonzero terms
    int capacity; // size of termArray
    int terms; // # of nonzero terms
```

Comparison

- What if a sparse polynomial?
  Example: $a(x)=2x^{1000}+1$
  - Representation 2: 1002 units of space
    - *a.degree*: 1
    - *coef*: 1001
  - Representation 3: 6 units of space
    - *a.capacity*: 1
    - *a.terms*: 1
    - *coef*: 2
    - *exp*: 2
- What if a dense polynomial?
  - Representation 3 uses about twice as much space as Representation 2 does
    - *exp* is implicit in Representation 2

Polynomial Addition

```
Polynomial Polynomial::Add(Polynomial b)
{// Return the sum of *this and b
    Polynomial c; int aPos = 0; int bPos = 0;
    while ((aPos < terms) && (bPos < b.terms))
        if (termArray[aPos].exp==b.termArray[bPos].exp) {
            float t = termArray[aPos].coef + termArray[bPos].coef;
            if (t) c.NewTerm(t, termArray[aPos].exp);
            aPos++; bPos++;
        } else if (termArray[aPos].exp<b.termArray[bPos].exp) {
            c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
            bPos++;
        } else {
            c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
            aPos++;
        } // end of if and while
    // add in remaining terms of *this
    for (; aPos < terms; aPos++)
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
    // add in remaining terms of b
    for (; bPos < b.terms; bPos++)
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
    return c;
} // end of Add
```

Merge terms

Copy ***this** or *b* to *c*

Time complexity: O(m+n)
-- m / n: # of nonzero terms in ***this** / b
e.g., $a(x) = 3x^2 + 2x - 4$
$b(x) = x^4 - 3x^2 + x + 3$

Adding a New Term

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{ // Add a new term to the end of termArray
    if (terms == capacity)
    { // double capacity of termArray
        capacity *= 2;
        term *temp = new term [capacity];           // new array
        copy(termArray, termArray + terms, temp);
        delete [] termArray;                        // deallocate old memory
        termArray = temp;
    }
    termArray[terms].coef = theCoeff;
    termArray[terms++].exp = theExp;
} // end of NewTerm
```

1. Enlarge size by 1 every “NewTerm”: time complexity: $O((m+n)^2)$
2. Double size if necessary: time complexity: $O(m+n)$

- Double only when *c.terms* == 1, 2, 4, 8, …
- If *c.terms* == *c.capacity* (= x), need $O(x)$
- $O(\sum_{i=0..k} 2^i) = O(2^{k+1}-1) = O(2^k) = O(2^{\lg(c.terms)}) = O(c.terms) = O(m+n)$

Array Applications: Sparse Matrices

Sparse Matrix

Dense

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}$$

- ***m* x *n* matrix: *m* rows, *n* cols**
  - 5 x 3
- **Dense: many nonzero terms**
  - 15/15
- **Store a matrix in a 2D array**
  - *a*[5][3]

Sparse

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

- **Sparse: many zero terms**
  - 8/36
- **What if a 5000 x 5000 matrix with only 5000 nonzero terms?**

ADT *SparseMatrix*

```
class SparseMatrix {
// A set of triples, <row, column, value>,
// where row and column are non-negative integers and form a unique combination;
// value is an integer.
public:
    SparseMatrix(int r, int c, int t);
    // the constructor function creates a SparseMatrix with
    // r rows, c columns, and a capacity of t nonzero terms

    SparseMatrix Transpose();
    // returns the SparseMatrix obtained by swapping the row and column value of each triple in *this

    SparseMatrix Add(SparseMatrix b);
    // if the dimensions of *this and b are the same, then the matrix produced by
    // adding corresponding items, namely those with identical row and column values is returned;
    // otherwise, error.

    SparseMatrix Multiply(SparseMatrix b);
    // if # of columns in *this equals # of rows in b
    // then the matrix d produced by multiplying *this by b according to the formula
    // d[i][j] = Σ(a[i][k] . b[k][j]), where d[i][j] is the (i, j)th element, is returned.
    // k ranges from 0 to the # of columns in *this – 1;
    // otherwise, error
};
```

SparseMatrix Representation

Representation

```
class SparseMatrix; // forward declaration

class MatrixTerm {
friend class SparseMatrix
private:
   int row, col, value;
};

in class SparseMatrix:
…
private:
     int rows, cols, terms, capacity;
     MatrixTerm *smArray;
```

Use triple <*row, col, value*>

Store triples row by row and within rows by columns (in ascending order)

Example

$$\begin{bmatrix} \boxed{15} & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

	row	***col***	***value***
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

Matrix Transpose

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
SparseMatrix SparseMatrix::Transpose()
{ // return the transpose of *this
    SparseMatrix b (cols, rows, terms); // capacity of b.smArray is terms
    if (terms > 0) { // nonzero matrix
        int CurrentB = 0; // the next position in b of the inserted term
        for (int c = 0; c < cols; c++) // transpose by columns; find the terms in 1st col and write to b
            for (int i = 0; i < terms; i++) // find & move terms in column c
                if (smArray[i].col == c) {
                    b.smArray[CurrentB].row = c;
                    b.smArray[CurrentB].col = smArray[i].row;
                    b.smArray[CurrentB].value = smArray[i].value;
                    CurrentB++;
                }
    } // end of if (terms > 0)
    return b;
} // end of Transpose
```

	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

$\Rightarrow$

	row	col	value
smArray[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	2	1	3
[4]	2	5	28
[5]	3	0	22
[6]	3	2	-6
[7]	5	0	-15

1. 2D array (*a*[][]): time complexity: O(*rows*cols*)
2. Triple (*smArray*): time complexity: O(*terms*cols*)
   $\Rightarrow$O(*rows*cols*^{2}) for dense matrix

Fast Matrix Transpose

```
SparseMatrix SparseMatrix::FastTranspose()
{ // return the transpose of *this in O(terms + cols) time.
    SparseMatrix b (cols, rows, terms);
    if (terms > 0) { // nonzero matrix
        int *rowSize = new int[cols];
        int *rowStart = new int[cols];
        // compute rowSize[i] = # of terms in row i of b
        fill(rowSize, rowSize + cols, 0); // initialize
        for (i = 0; i < terms; i++) rowSize[smArray[i].col]++;
        // rowStart[i] = starting position of row i in b
        rowStart[0] = 0;
        for (i = 1; i < cols; i++)  rowStart[i] = rowStart[i-1] + rowSize[i-1];
        for (i = 0; i < terms; i++) { // copy from *this to b
            int j = rowStart[smArray[i].col];
            b.smArray[j].row = smArray[i].col;
            b.smArray[j].col = smArray[i].row;
            b.smArray[j].value = smArray[i].value;
            rowStart[smArray[i].col]++;
        } // end of for
        delete [] rowSize;
        delete [] rowStart;
    } // end of if
    return b;
} // end of FastTranspose
```

O(*terms*)

O(*cols*)

O(*terms*)

Time complexity: O(*cols*+*terms*)
⇒O(*rows***cols*) for dense matrix

*this	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

b	row	col	value
smArray[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	2	1	3
[4]	2	5	28
[5]	3	0	22
[6]	3	2	-6
[7]	5	0	-15

rowSize[0] = 2 *rowStart*[0] → [0]

rowSize[1] = 1 *rowStart*[1] → [2]

rowSize[2] = 2 *rowStart*[2] → [3]

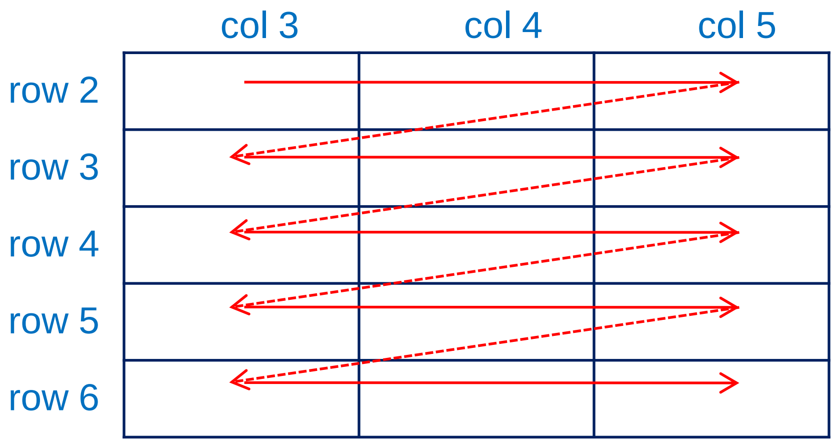
rowSize[3] = 2 *rowStart*[3] → [5]

rowSize[5] = 1 *rowStart*[5] → [7]

Representations of Arrays

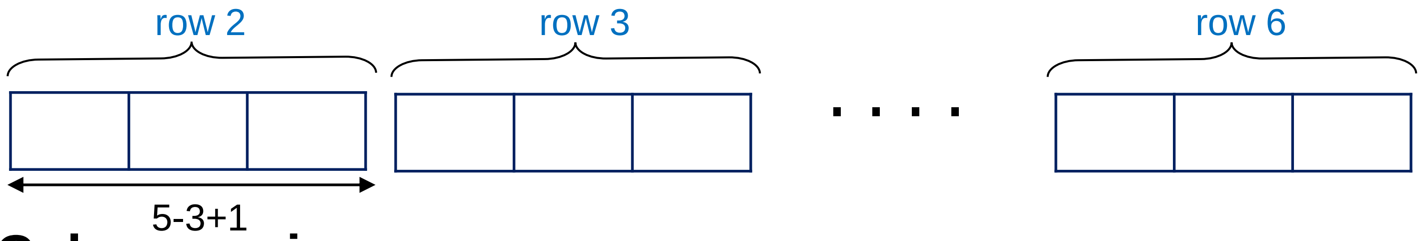
Two-Dimensional Arrays

- ***n*=2, declare *a*[2:6, 3:5]**



# of elements
= (6-2+1)*(5-3+1) = 5*3 = 15

- **Row major:**
  - lexicographic order: *a*[2][3], *a*[2][4], *a*[2][5], *a*[3][3], *a*[3][4], …



- **Column major**
  - *a*[2][3], *a*[3][3], …, *a*[6][3], *a*[2][4], … …., *a*[6][5]

Multidimensional Arrays

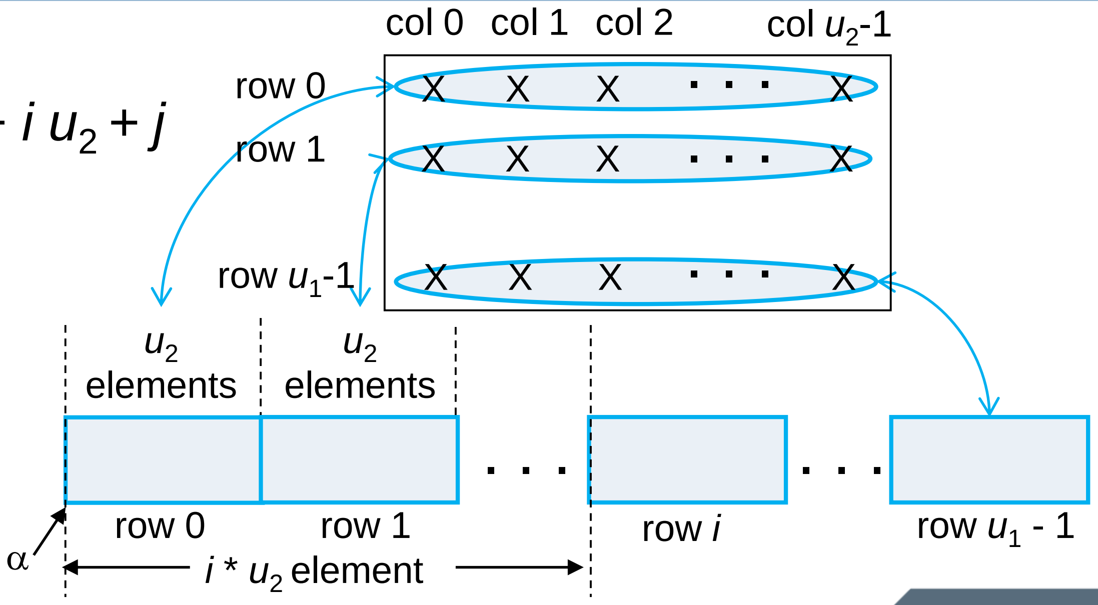
- **Implement an n-dimensional array into a one-dimensional array via either row major order or column major order**
  - In row major order, declare a C++ array $a[u_1][u_2]\ldots[u_n]$ starting at the address α, what is the address of $a[i_1][i_2]\ldots[i_n]$?

$$
\begin{aligned}
&= \alpha + i_1 u_2 u_3 \ldots u_n \\
&\qquad + i_2 u_3 u_4 \ldots u_n \\
&\qquad + i_3 u_4 u_5 \ldots u_n \\
&\qquad . \\
&\qquad . \\
&\qquad + i_{n-1} u_n \\
&\qquad + i_n
\end{aligned}
$$

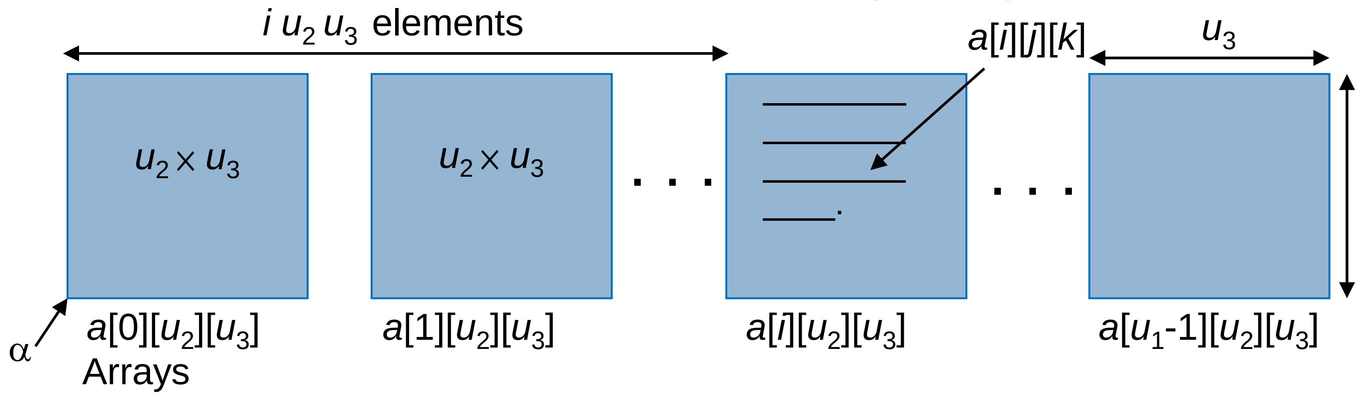
$= \alpha + \sum_{j=1..n} i_j a_j$, where $a_j = \Pi_{k=j+1..n} u_k$, $1 \le j < n$, and $a_n = 1$

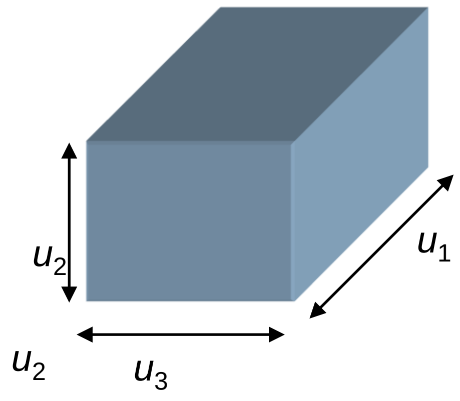
Example: Two & Three-Dimensional Arrays

- $n = 2$, $a[u_1][u_2]$
- Address of $a[i][j] = \alpha + i\, u_2 + j$



- $n = 3$, $a[u_1][u_2][u_3]$
- Address of $a[i][j][k] = \alpha + i\, u_2\, u_3 + j\, u_3 + k$





Strings

ADT *String*

H	e	l	l	o		W	o	r	l	d	\0

class *String* **{**
// objects: an ordered set of zero or more characters, $S = s_0, \ldots, s_{n-1}$. If $n = 0$, S is an empty/null string.

public:
    String(**char** **init*, **int** *m*);
    // constructor that initializes ***this** to string *init* of length *m*

    bool operator==(*String t*);
    // if the string represented by ***this** equals *t*, return true;
    // else return false

    bool operator!();
    // if ***this** is empty then return true; else return false

    int *Length*();
    // return the # characters in ***this**

    String Concat(*String t*);
    // return a string composed by *this followed by *t*

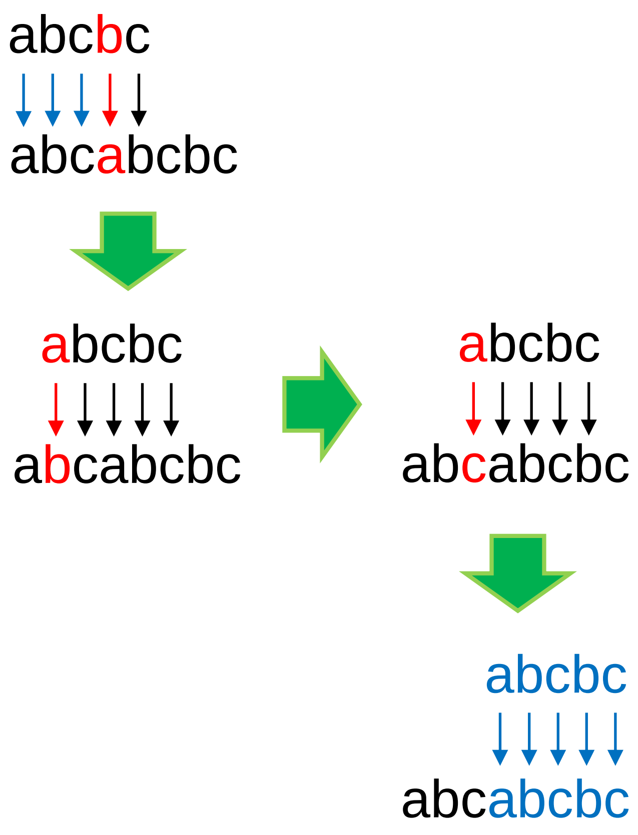
    String Substr(**int** *i*, **int** *j*);
    // return the substring starting at *i* with length *j*

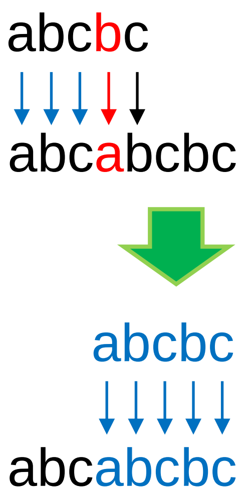
    int *Find*(*String pat*);
    // return an index *i* if *pat* matches the substring of ***this** starting at *i*; return -1 if not found or *pat* null
};

String Pattern Matching

Exhaustive: O(*lengthP*lengthS*)

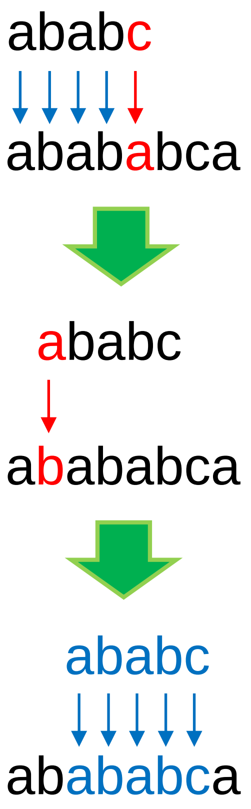
Intelligent: O(*lengthP+lengthS*)?



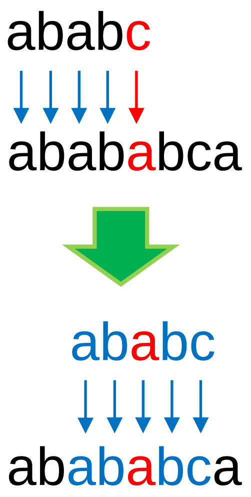


String Pattern Matching

Exhaustive: O(*lengthP*lengthS*)**



Intelligent: O(*lengthP*+*lengthS*)?

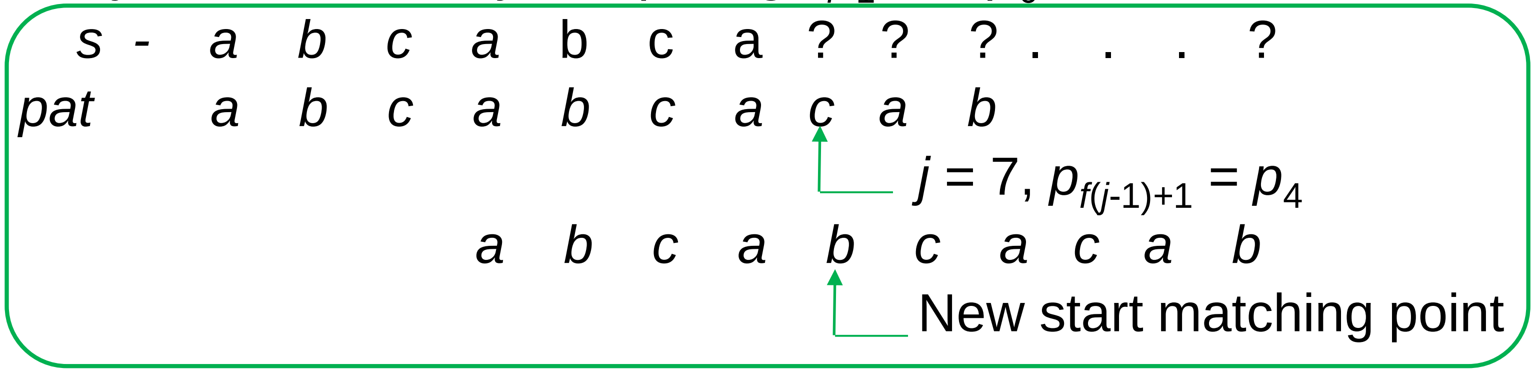


The Knuth-Morris-Pratt Algorithm (1/3)

- **Definition: the failure function f of a pattern $p = p_0 p_1 \ldots p_{n-1}$ is**
  - $f(j)$ = largest $k < j$, s.t. $p_0 p_1 \ldots p_k = p_{j-k} p_{j-k+1} \ldots p_j$ if $k \geq 0$ exists;
    -1, otherwise.

j	0	1	2	3	4	5	6	7	8	9
pat	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
f	-1	-1	-1	0	1	2	3	-1	0	1

- **Can we avoid resuming matching from scratch?**
  - If a partial match is found s.t. $s_{i-j} \ldots s_{i-1} = p_0 p_1 \ldots p_{j-1}$ and $s_i \neq p_j$
    - if $j \neq 0$, resume matching by comparing s_i and $p_{f(j-1)+1}$
    - if $j = 0$, continue by comparing s_{i+1} and p_0



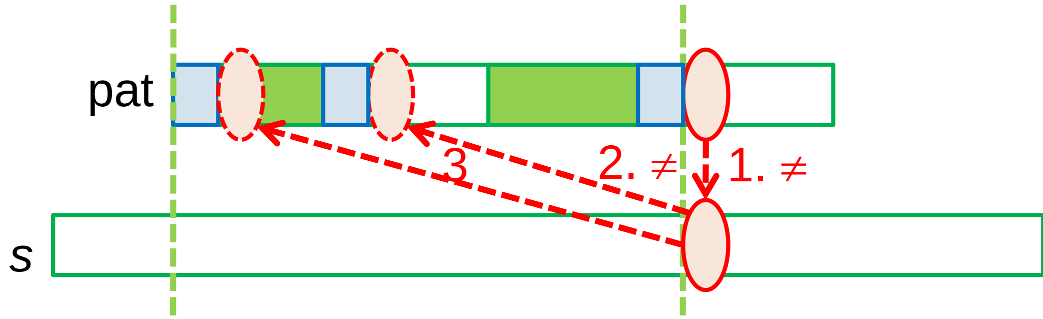
The Knuth-Morris-Pratt Algorithm (2/3)

```
int String::FastFind(String pat) {
// Determine if pat is a substring of s
    int posP = 0, posS = 0;
    int lengthP = pat.Length(), lengthS = Length();

    while ((posP<lengthP)&&(posS<lengthS)) {
        if (pat.str[posP] == str[posS]) { // char match
            posP++, posS++;
        } else // unmatched
            if (posP == 0) posS++; // if unmatched and j = 0, compare s_{i+1} and p_0
            else posP = pat.f[posP-1]+1; // if unmatched and j ≠ 0, compare s_i and p_{f(j–1)+1}
    }

    if (posP<lengthP) return -1; // not found
    else return posS-lengthP; // matched!
} // end of FastFind
```

FastFind:
time complexity: $O(lengthS)$



The Knuth-Morris-Pratt Algorithm (3/3)

```
void String::fail() {
// compute the failure func for pattern *this
    int lengthP=Length();
    f[0]=-1;
    for (int j =1; j<lengthP; j++) { // compute f[j]
        int i = f[j-1];
        while ((*(str+j)!=*(str+i+1))&&(i>=0)) i=f[i];
        if (*(str+j)==*(str+i+1)) f[j]=i+1;
        else f[j]=-1;
    } // end of for
} // end of fail
```

fail: preprocessing:
time complexity: $O(lengthP)$

KMP:
time complexity: $O(lengthP+lengthS)$

$f(j) = -1$, if $j=0$;
$f^m(j-1)+1$, m is the least int k s.t. $p_{f^k(j-1)+1}=p_j$;
-1, otherwise.

Note:
$f^1(j) = f(j)$, $f^m(j) = f(f^{m-1}(j))$