2. Arrays and Structures

h. choi

hchoi@handong.edu

Agenda

- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- Representation of Multidimensional Arrays
- Ordered Lists → Not in text
- Strings

What is an Array?

- Conventional definition of array
  - “A consecutive set of memory location”

A[0]	A[1]	A[2]	…	A[n-1]

< Array A of size n >

- Easy, but it’s based on perspective of implementation

→ Let’s try to find deeper understanding about array

- “What is the essence of the array?”

→ Let’s think about an alternative definition as ADT.

  - Separate implementation details from definition of array.

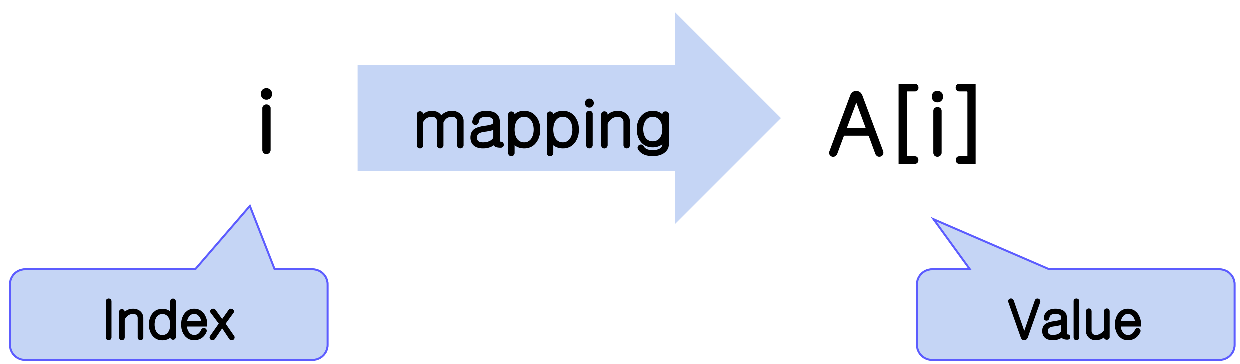
Array As an ADT

- Abstract Data Type *Array*
  - Objects: a set of pairs <*index*, *value*>
    - Each value of index is mapped with a value from the *set item*
    - *Index*: finite ordered set of one or more dim.

      Ex) {0, …, *n*-1}: 1D index

      {(0,0), (0,1), …, (*n*-1, *m*-1)} : 2D index

"type" of elements



Operations of Array

- Create
  - Creates an array given its size
- Retrieve
  - Retrieve an element given its index
- Store
  - Store an element given its index and content
- Destroy
  - Destroy the array

Array As an ADT

- Abstract Data Type *Array*
  - Operations

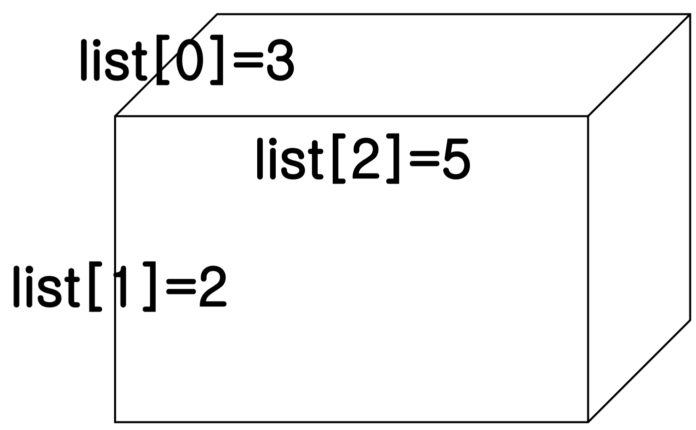
    For $A \in Array$, $i \in index$, $x \in item$ and j, $size \in$ integer

    - ***Array*** **Create(j, list)**

      ::= return an *Array* of *j* dims, where list is a *j*-tuple whose *k*-th element is the size of *k*-th dim. Items are undefined.

$$\text{int A}\overbrace{[3][2][5]}^{j\,=\,3\ \text{(dim.)}};$$

list[0]: 3 (depth)	
list[1]: 2 (height)	
list[2]: 5 (width)	



< 3D array >

Array As an ADT

- Abstract Data Type *Array*
  - Operations (cont.)

    For $A \in$ *Array*, $i \in$ *index*, $x \in$ *item* and j, *size* $\in$ integer

    - **Item Retrieve(*A*, *i*)**

      ::= if($i \in$ *index*), return item associated with *i* in *Array* A
      else return error

      value = A[i];

    - ***Array* Store(*A*, *i*, *x*)**

      ::= if($i \in$ *index*), return an array that is identical to *Array* A except the new pair <*i*, *x*> has been inserted
      else return error.

      A[i] = x; is equivalent to “A = Store(A, i, x);”

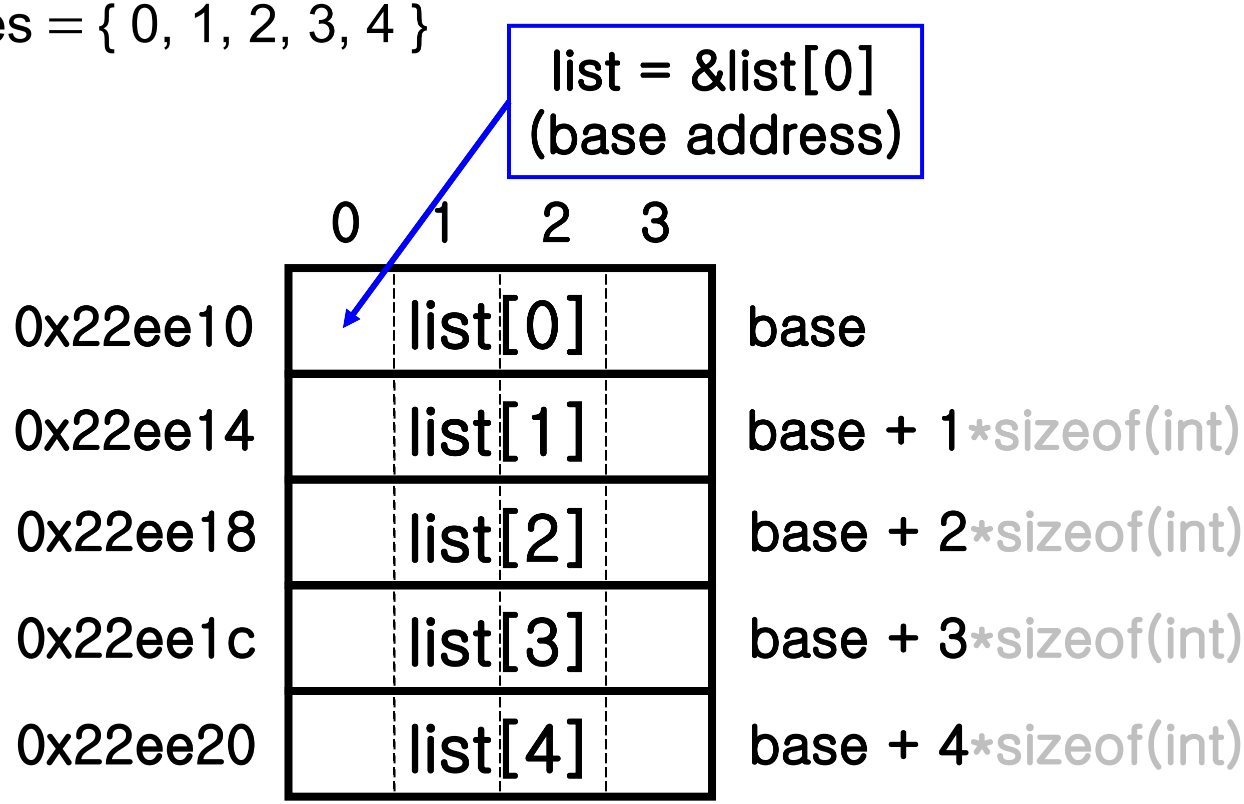
Array in C Language

- Can be viewed as an implementation of ADT array

  Ex) int list[5]; // integer array containing 5 elements

  int *plist[5]; // pointer array containing 5 elements

  - Indices = { 0, 1, 2, 3, 4 }



Array in C Language

- Addition/subtraction operation on pointer implies multiplication by sizeof(type)
  - for **int** *p,
    if p == 0x22ee10,
    then p+i == 0x22ee10 + i * sizeof(int)
  - A[i] == *(A+i);   // In C, array is implemented by pointer

cf. How can we access *p* + *i* (in bytes)?,

int *pp = (int *)((**char** *)p + i);

Note! **(char *)p + i = 0x22ee10 + i**, because sizeof(char) = 1

Array and Pointer

- size information of array is contained in array.

```
void function(int array_arg[])
{
   int noEntry = sizeof(array_arg) / sizeof(int);      //we don't know the size
   printf("(%d, %d)\n", sizeof(array_arg), noEntry);
}

void main()
{
   int array[5];
   int noEntry = sizeof(array) / sizeof(int);

   printf("(%d, %d)\n", sizeof(array), noEntry);

   function (array);
}
```

practice

Agenda

- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- Representation of Multidimensional Arrays
- Ordered Lists → Not in text
- Strings

One-Dimensional Arrays

- Statically allocated array vs. dynamically allocated array
  - int list[100];
  - int *list = (int*)malloc(100 * sizeof(int));
    - Later list should be deallocated by "free(list);"

  - After allocation, its use is almost the same as the statically allocated array.

```
Ex) for(i = 0; i < 100; i++)
            list[i] = i;
```

calloc
allocates bytes
and initializes them to 0

- Why dynamically allocated array?
  - **The size of array is decided at runtime.**

Allocating 1D Array

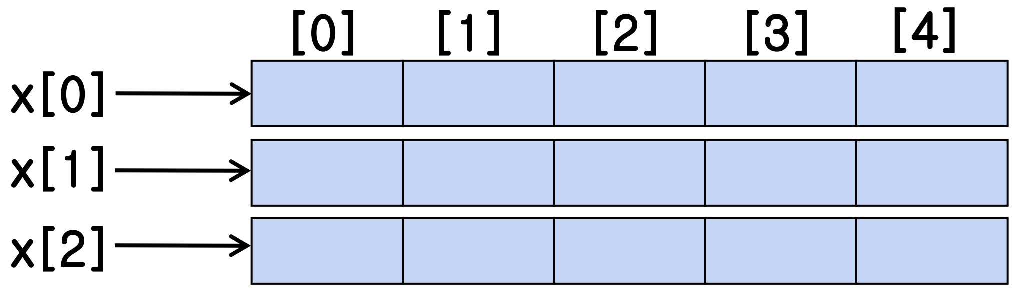
Ex) Reading a series of numbers from the user

```
int n = 0, *list = NULL;
printf("Enter the number of integers to read: ");
scanf("%d", &n");
if (n < 1) {
    printf( "Improper value of n. \n");
    exit(-1);
}

list = malloc(n * sizeof(int));
if(list == NULL){
    printf( "Failed to allocate memory.\n");
    exit(-1);
}
```

Two-Dimensional Arrays

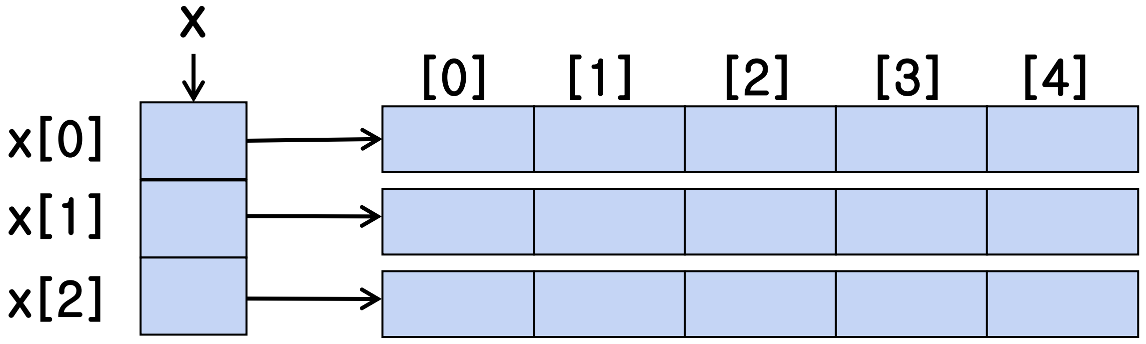
- 2D is represented as a 1D array, where each element is a 1D array

  Ex) int x[3][5];



- Dynamically allocated 2D array

4 memory blocks



Allocating 2D Arrays

- Function to allocate 2D array

```
int ** make2dArray(int rows, int cols)
{ /* create a two dimensional rows × cols array */
    int **x = NULL, i = 0;

    x = malloc(rows * sizeof (*x)); /* get memory for row pointers */

    for (i = 0; i < rows; i++)      /* get memory for each row */
      x[i] = malloc(cols * sizeof(**x));

    return x;
}
```

practice

- Usage

```
int **myArray = NULL;
myArray = make2dArray(3,5);
myArray[1][2] = 6;
```

Deallocating 2D Arrays

- Function to deallocate 2D array

```
void free2dArray(int **array, int rows)
// deallocates a two dimensional array
{
    int i = 0;

    for (i = 0; i < rows; i++) /* free memory for each row */
        free(array[i]);
    free(array);               /* free memory for row pointers */

    return;
}
```

practice

- Usage

```
int **myArray = NULL;
myArray = make2dArray(3,5);
myArray[1][2] = 6;
…
free2dArray(myArray, 5);
```

Agenda

- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- Representation of Multidimensional Arrays
- Ordered Lists → Not in text
- Strings

Structures

- **Struct**: collection of data items, each item may differ in type

```
struct tPerson {        // tag
    char name[10];      // name in char array
    int age;            // age in integer
    float salary;       // salary of person in float
} person1;              // instance name

struct tPerson person2;
    // equivalent to the previous declaration of "person1"
```

- Access to member: '.' (member operator), "->" operator

  Ex) person1.name, person1.age, person1.salary

Why Structures ?

- Structures explicitly represent the relation among attributes.

  Ex) Representation of 20 persons

  - Using arrays

```
char name[20][10];
int age[20];
float salary[20];
```

→ **Difficult to understand relation between attributes**

  - Using structure

```
struct  tPerson {
   char name[10];
   int age;
   float salary;
} person[20];
```

→ **Easy to understand**

name[k], age[k], salary[k]

vs.

person[k]

Structures

- Struct can be defined as a type by *typedef*

Ex) ***typedef*** struct { // tag was omitted

// members

} human_being;

type name

typedef struct tPerson *human_being;*

type name

tag

typedef struct tPerson{

// members

} human_being ;

tag

type name

To make instances

```c
struct tPerson {
  char name[10];
  int age;
} person1;
```

```c
struct tPerson {
  char name[10];
  int age;
};
struct tPerson person1;
```

```c
typedef struct {
  char name[10];
  int age;
} Tperson;
Tperson person1;
```

```c
typedef struct tPerson Tperson;
struct tPerson {
  char name[10];
  int age;
};
Tperson person1;
```

Structures

- Comparison of structure variables is NOT defined in C language

  Ex) struct Person person1, person2;

  printf("person1 == person2 = %d\n", person1 == person2);

  // compile error occurs

  cf) Comparison of array is defined as comparison of pointer

- **Assignment of structure is provided in ANSI C**

  Ex) struct Person person1, person2;

  …

  person2 = person1;   // all members are copied

  - But not available in old-style C

  cf) Assignment of array is NOT allowed.

Comparison of Structure Variables

- A function to compare two instances of human_being structure

```
int humansEqual(human_being person1, human_being person2)
{
  if(strcmp(person1.name, person2.name))
    return FALSE;
  if(person1.age != person2.age)
    return FALSE;
  if(person1.salary != person2.salary)
    return FALSE;

  return TRUE;
}
```

Structures as Parameters

- Passing struct as parameter

```
struct MyStruct {
    char *name;
    int age;
    // …
};

int main()
{
    struct MyStruct a;
    Print(a);
    PrintPtr(&a);
}
```

```
void Print(struct MyStruct a)
{              // assignment of structure
    printf("name = %s\n", a.name);
    …
}

void PrintPtr(struct MyStruct *p)
{
    printf("name = %s\n", p->name);
    …
}
```

→ more efficient
Why?

Embedded Structures

- Embedded structure

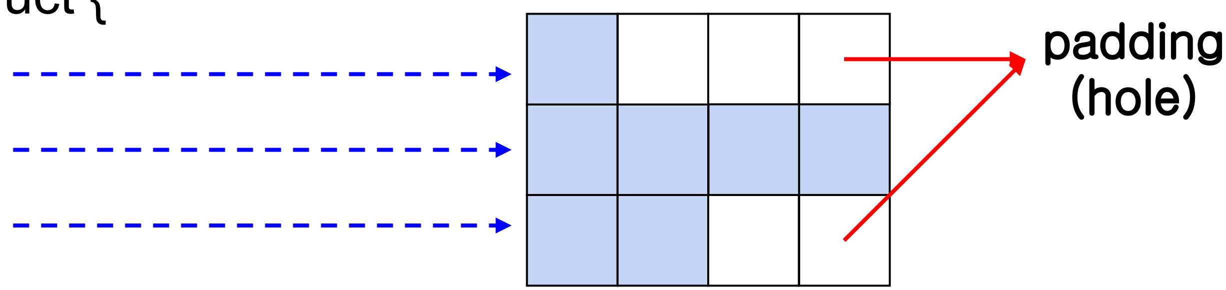
```
typedef struct {
      int month;
      int day;
      int year;
} date;

typedef struct {
      char name[10];
      …
      date dob;  // date of birth
} human_being;

Ex) human_being john;
       john.dob.year = 1971;
       john.dob.month = 4;
       john.dob.day = 15;
```

Structures

- Internal representation of structure
  - Padding (or hole) can be included for memory alignment
    - To improve access speed
    - sizeof(struct) != Σ sizeof(members)
  - Memory alignment often improves efficiency on many CPU's.

Ex)

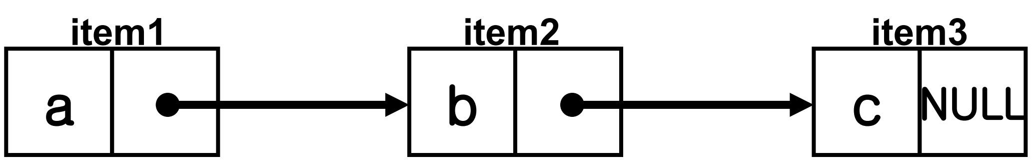
```
struct MyStruct {
    char ch;
    int i;
    short s;
}
```



Structures

- Self-referential structures

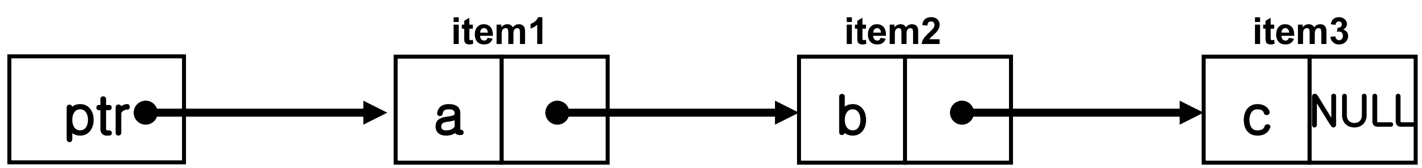
```
typedef struct tList {
   char data;
   struct tList *link;              // pointer to the same type
} List;
List item1, item2, item3;
item1.data = 'a';     item2.data = 'b';     item3.data = 'c';
item1.link = &item2;
item2.link = &item3;
item3.link = NULL;
```



Structures

- Self-referential structures

```
List *ptr;

ptr = &item1;
while (ptr){
    printf("%c\n", ptr->data);
    ptr = ptr->link;
}
```



Contacts with structure

- read and parse lines from a file.
  - each line of files is in the following format.
    - name; birthday; email; phone_number
    - ex) henry choi; 20190303; hchoi@handong.edu; 010-1234-5678

- add the data into array of structure named 'Contact'

practice

Agenda

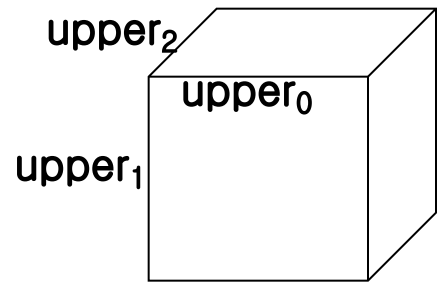
- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- <u>Representation of Multidimensional Arrays</u>
- Ordered Lists → Not in text
- Strings

Multidimensional Arrays

- Multidimensional array

  $A[upper_0][upper_1][upper_2]$

  - Total size = $\Pi upper_i$



- Internal representation: multidimensional array are <u>serialized into 1D memory</u>

  Ex) 2D array A[m][n]

$$
m\left\{\overbrace{\begin{bmatrix} A[0][0], A[0][1], \ldots, A[0][n-1] \\ A[1][0], A[1][1], \ldots, A[1][n-1] \\ \ldots \quad , A[i][j], \quad \ldots \\ A[m-1][0], A[m-1][1], \ldots, A[m-1][n-1] \end{bmatrix}}^{n}\right.
$$

< 2D array >

A[0][0],A[0][1],…, A[0][n−1], A[1][0], A[1][1], …, A[1][n−1],…, A[m−1][0], A[m−1][1],…, A[m−1][n−1]

< Memory >

Multidimensional Arrays

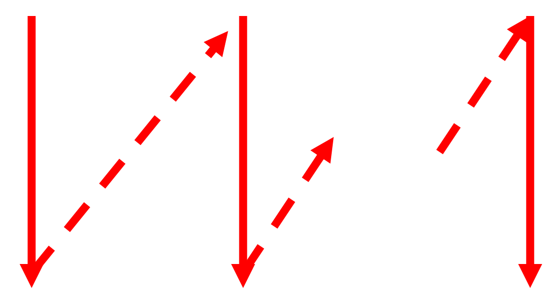
- Representation of 2D array
  - **<u>Row-major format</u>**
    - A[i][j] ≡ *((int*)A + i * n + j)

$$\begin{bmatrix} A[0][0], A[0][1], \ldots, A[0][n-1] \\ A[1][0], A[1][1], \ldots, A[1][n-1] \\ \ldots \\ A[m-1][0], A[m-1][1], \ldots, A[m-1][n-1] \end{bmatrix}$$

< row−major format >

  - Column-major format (less popular)
    - A[i][j] ≡ *((int*)A + i + j * m)

$$\begin{bmatrix} A[0][0], A[0][1], \ldots, A[0][n-1] \\ A[1][0], A[1][1], \ldots, A[1][n-1] \\ \ldots \\ A[m-1][0], A[m-1][1], \ldots, A[m-1][n-1] \end{bmatrix}$$



< column−major format >

Example

- Test program

```
const int rows = 3 , cols = 5;
int array2D[rows][cols];
int *array1D = (int *)array2D;
int i = 0, j = 0, n = 0;

for(i = 0; i < rows; i++)
    for(j = 0; j < cols; j++)
        array2D[i][j] = n++;

for(i = 0; i < rows; i++)
    for(j = 0; j < cols; j++)
        printf("array2D[%d][%d](%p) = %d, array1D[%d](%p) = %d\n",
         i, j, &array2D[i][j], array2D[i][j],
         i*cols+j, &array1D[i*cols+j], array1D[i*cols+j]);
```

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

Example

- Result

```
array2D[0][0](0x22ede0) = 0, array1D[0](0x22ede0) = 0
array2D[0][1](0x22ede4) = 1, array1D[1](0x22ede4) = 1
array2D[0][2](0x22ede8) = 2, array1D[2](0x22ede8) = 2
array2D[0][3](0x22edec) = 3, array1D[3](0x22edec) = 3
array2D[0][4](0x22edf0) = 4, array1D[4](0x22edf0) = 4
array2D[1][0](0x22edf4) = 5, array1D[5](0x22edf4) = 5
array2D[1][1](0x22edf8) = 6, array1D[6](0x22edf8) = 6
array2D[1][2](0x22edfc) = 7, array1D[7](0x22edfc) = 7
array2D[1][3](0x22ee00) = 8, array1D[8](0x22ee00) = 8
array2D[1][4](0x22ee04) = 9, array1D[9](0x22ee04) = 9
array2D[2][0](0x22ee08) = 10, array1D[10](0x22ee08) = 10
array2D[2][1](0x22ee0c) = 11, array1D[11](0x22ee0c) = 11
array2D[2][2](0x22ee10) = 12, array1D[12](0x22ee10) = 12
array2D[2][3](0x22ee14) = 13, array1D[13](0x22ee14) = 13
array2D[2][4](0x22ee18) = 14, array1D[14](0x22ee18) = 14
```

Multidimensional Arrays

- Generalized row-major format
  - int A[$upper_0$] [$upper_1$]…[$upper_{n-1}$]
    - A[i_0] [i_1]…[i_{n-1}] = *((int*)A + i_0*$upper_1$*$upper_2$…$upper_{n-1}$
      + i_1*$upper_2$…$upper_{n-1}$
      + i_2*$upper_3$…$upper_{n-1}$
      + i_{n-2}*$upper_{n-1}$
      + i_{n-1})

$$= *(A + \sum_{j=0}^{n-1} i_j a_j) \quad \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k, \text{ for } 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

Efficient Access to 2D Array [sup]

- Efficient way

```
int matrix[numRows][numColumns];
for ( row = 0; row < numRows; row++ ) {
  for ( column = 0; column < numColumns; column++ ){
    matrix[ row ][ column ] = 0;
  }
}
```

assuming row major format

- Inefficient way

```
for ( column = 0; column < numColumns; column++ ){
  for ( row = 0; row < numRows; row++ ) {
    matrix[ row ][ column ] = 0;
  }
}
```

Efficient Access to 2D Array [sup]

practice
check the execution time

- 1D implementation
  ```
  int matrix1D[numRows*numColumns];
  for (i = 0; i < numRows*numColumns; i++ )
      matrix1D[i] = 0;
  ```

  → About 11% faster than 2D array initialization in C++
  → About 47% faster than 2D array initialization in Java
  High-dimensional arrays are usually more expensive than low-dimensional array

- More optimized version (?): pointer operation
  ```
  int *pLimit = matrix1D + numRows*numColumns;
  int *p;
  for(p = matrix1D; p < pLimit; p++)
      *p = 0;
  ```

  → In some environment, this optimization provides little improvement in efficiency.

Agenda

- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- Representation of Multidimensional Arrays
- <u>Ordered Lists</u> → Not in text
- Strings

Ordered List [Aho, Hopcroft, Ullman]

- Ordered list: a sequence of zero or more elements of a given type

$$a_0, a_1, a_2, \ldots, a_{n-1}$$

    - n: length of list
  - Important property: all elements are **linearly ordered** according to its 'position'
    - Implementation of 'position' can vary

Ex) Student attendance roll (ordered by student number)

  - Implementation: **array**, linked list, cursor, …

Ordered List

- Operations on ordered list

  For $L \in$ list, $x \in$ element, $p \in$ position

  - **Insert(*L*, *x*, *p*)** ::= insert *x* at position *p* in list *L*

    Ex) **Insert(<a,b,c,d,e>, n, 3) → L = <a,b,c,n,d,e>**

  - **Delete(*L*, *p*)** ::= delete element at position *p* on list *L*

    Ex) **Delete(<a,b,c,d,e>, 2) → L = <a,b,d,e>**

  - **MakeNull(*L*)** ::= make *L* an empty list
  - **Locate(*L*, *x*)** ::= return position of *x* on list *L*

    if *x* does not exist on *L*, return *error*
  - **Retrieve(*L*, *p*)** ::= return element at position *p* on list *L*

Ordered List

- Operations on ordered list

  For $L \in$ list, $x \in$ element, $p \in$ position

  - **Next/Previous(*L*, *p*)** ::= return position of following/preceding position *p* on list *L*
  - **First(*L*)/Last(*L*)** ::= return position of first/last element
  - **End(*L*)** ::= return position of next to last element
    - <u>End(*L*) is just for boundary condition, and should not be used to access the element</u>

$$a_0, a_1, a_2, \ldots, a_{n-1}, a_n$$

First(L)

Last(L)

End(L)

```
for(p = First(L); p < End(L); p = Next(L, p)){
  ...
}
```

Array Implementation of List

- List elements are stored in contiguous cells of array
  - Position is represented by integer

a_0	a_1	a_2	…	a_{n-1}	empty
					size max_len

  - Data representation
    - Array ***elements*** to store elements, whose size is ***max_len***
    - Integer ***size*** to store # of elements
  - Operations
    - Implementations of most operations are straightforward.
    - For Insert/Delete, elements should be shifted.

Agenda

- Arrays
- Dynamically Allocated Arrays
- Structures
- (Polynomials)
- (Sparse Matrices)
- Representation of Multidimensional Arrays
- Ordered Lists → Not in text
- Strings

String

- String: concatenation of characters
  - $S = s_0 s_1 s_2 \ldots s_{n-1}$
    - s_i: characters from the character set of the programming language

Ex) char str[6] = “HELLO”; // in C

// str = ‘H’ + ‘E’ + ‘L’ + ‘L’ + ‘O’ + ‘\0’

Let’s define String by ADT

String ADT

- Abstract Data Type *String*
  - Objects: a finite sequence of zero or more characters
  - Functions

    s, $t \in$ *String*, i, j, $m \in$ non-negative integers

    - ***String* Null(*m*)** := return a string whose maximum length is *m*
      but initially set to null string("");
    - **int Compare(*s*, *t*)** := if *s* equals *t* return 0
      else if *s* precedes *t* return -1
      else return +1
    - **Boolean IsNull(*s*)** := *s* is a null string return TRUE
      else return FALSE
    - **int Length(*s*)** := return # of characters

String ADT

- Abstract Data Type *String*
  - Functions (cont.)

    $s, t \in String$, $i, j, m \in$ non-negative integers

    - ***String* Concat(*s*, *t*)** := if t is not a null string
      return a string whose elements are
      those of s followed by those of t
      else return s

      Ex) Concat("abc", "123") == "abc123"

    - ***String* Substr(*s*, *i*, *m*)** := if($i > 0$ && $(i+m-1) < \text{Length}(s)$)
      return string containing char's of s at
      positions $i, i+1, \ldots, i+m-1$

      Ex) Substr("ABC**DE**FG", 3, 2) = "DE"

String in C Language

- String is represented by character array terminated with null character ('\0') in C.

  Ex) char s[100] = "dog";

0	1	2	3	99
d	**o**	**g**	**\0**	**…**

char s[] = "dog";

d	**o**	**g**	**\0**

String Functions in C Language

Declared in *string.h*

- strcat, strncat : string concatenation
- strcmp, strncmp : string comparison
  - strcmp("ABC", "ABC") → 0
  - strcmp("ABC", "AB**B**") → 1
  - strcmp("ABC", "AB**D**") → -1
  - strncmp("Hello", "Hello, World", 5) → 0
- strcpy, strncpy : string copy
- strlen : string length

String Functions in C Language

- strchr, strrchr, strstr: find character or substring

  char s[] = “Hello, World”;

  - strchr(s, ‘l’) → s + 2 // search from left
  - strrchr(s, ‘l’) → s + 10 // search from right
- strtok(s, delim) : return token delimited by delim
  - strtok(s, “,”) → “Hello”
- strspn(s, spanset), strcspn, strpbrk :

  scan s for characters in (not in) spanset

  char s[] = “Hello, **World**”;

  - strspn(s, “Hle”) → 4 // length of “Hell”
  - strcspn(s, “WXYZ”) → 7 // length of “Hello, ” right before ’W’
  - strpbrk(s, “WXYZ”) → s + 7 // like strcspn, except pointer.

strtok

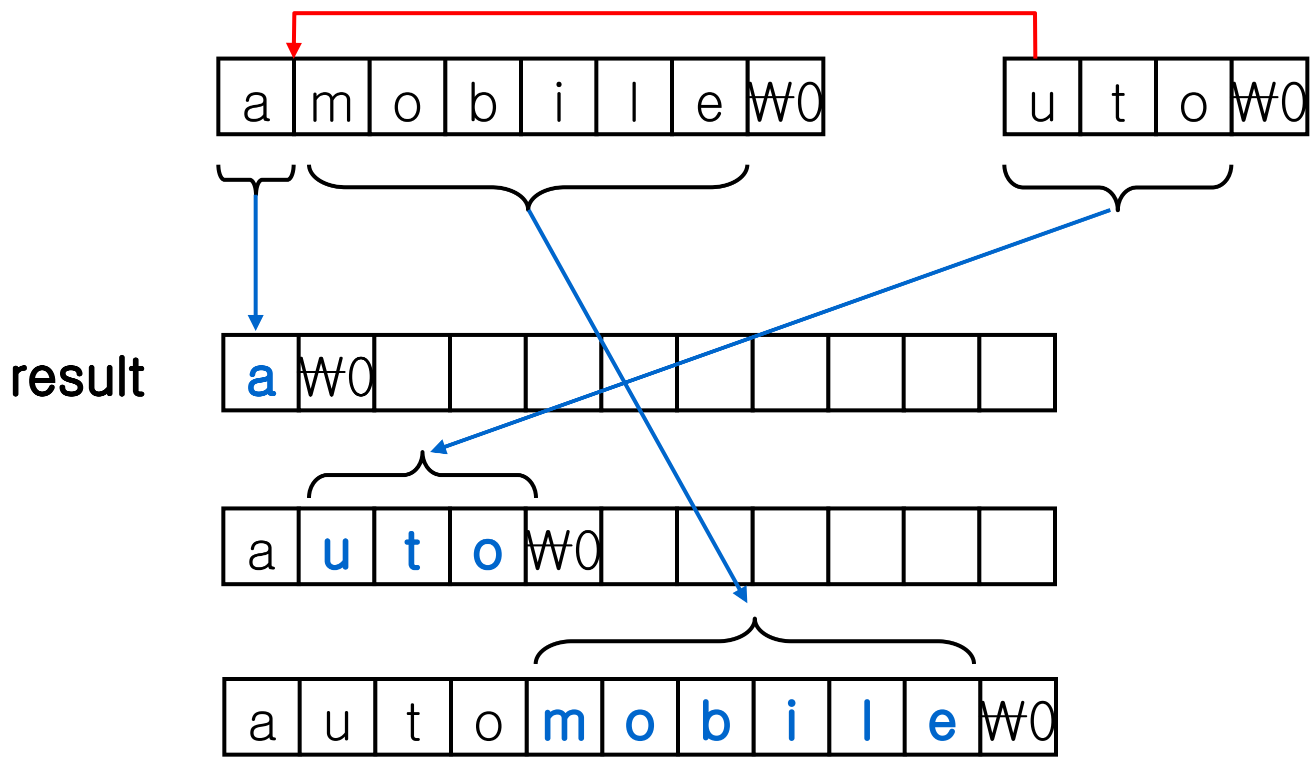
```
char array[] = “I love you, HGU”;
char *ptr = strtok(array," ");

while (ptr) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, " ");
}
```

output:

```
I
love
you,
HGU
```

String Insertion

Example) Given string1 = “amobile” and string2 = “uto”, insert string2 into position 1 of string1 to make “automobile”



String Copy Functions

- strcpy(*dest*, *src*): copy *src* to *dest*

  Ex)

  char src[] = “ABCDE”;

  char dest[100] = “0123456789”;

  strcpy(dest, src); // dest = “ABCDE”

- strncpy(*dest*, *src*, *n*): copy no more than *n* characters of *src* to *dest*.

  If *n* is equal or less than length of *src*,

  ‘\0’ is NOT appended to the result automatically.

  Ex)

  strncpy(dest, src, 3) // dest = “ABC3456789”;

String Concatenation Functions

- strcat(dest, src): append string *src* to the end of *dest*

  Ex)

  char src[] = “, World”;

  char dest[100] = “Hello”;

  strcat(dest, src);   // dest == “Hello, World”;

- strncat(dest, src, n): append no more than *n* characters of *src* to the end of *dest*. ‘\0’ is always appended.

  Ex)

  strncat(dest, src, 3); // dest == “Hello, W”

String Insertion

practice

```
void strnins(char *s, char *t, int i)
{
    char temp[100];
    //char* temp = (char*)malloc(strlen(s)+strlen(t)+1);

    if(strlen(s) == 0)
        strcpy(s, t);
    else if(strlen(t)){
        strncpy(temp, s, i); // \0 is not attached
        temp[i] = 0;        // don't forget
        strcat(temp, t);
        strcat(temp, s+i);
        strcpy(s, temp);
    }
    //free(temp);
}
```

String Pattern Matching

- Find a pattern from string
  - strstr(string, pattern) in standard C library

    if(pattern exists in string) return position of pattern in string

    else return NULL

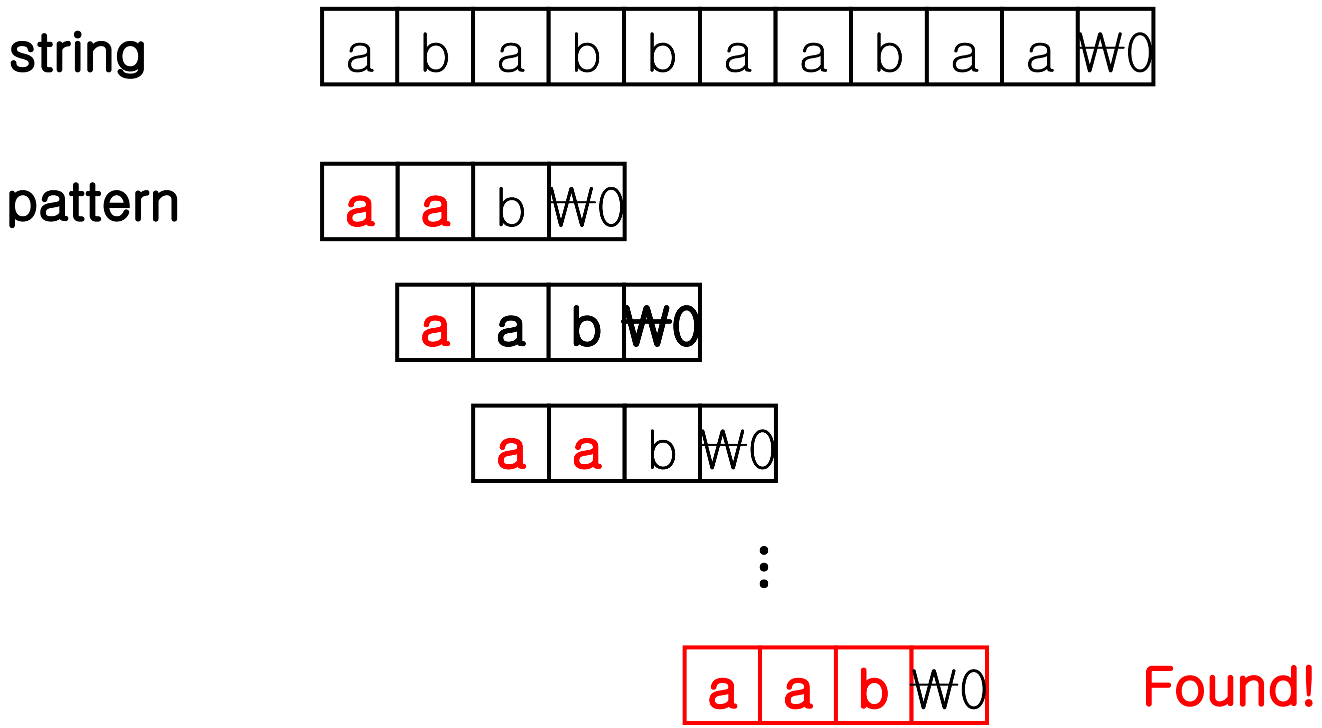
Ex) s = "ex**amp**le";

p = strstr(s, "amp") → s + 2;

index = p – s;

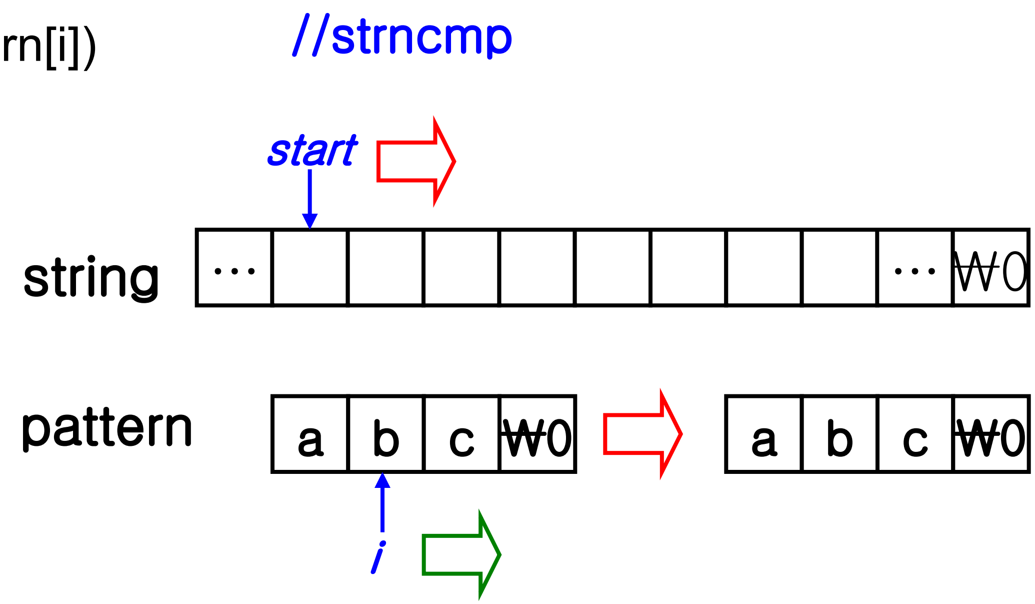
Simple Implementation

- Exhaustive matching



Simple Implementation

```
int pattern_matching(char *string, char *pattern)
{
   int start, i;
   int lens = strlen(string);
   int lenp = strlen(pattern);

   for(start = 0; start + lenp <= lens; start++){
      for(i = 0; i < lenp; i++){
         if(string[start+i] != pattern[i])
            break;
      }
      if(i == lenp) //found
          return start;
   }
   return -1; // not found
}
```

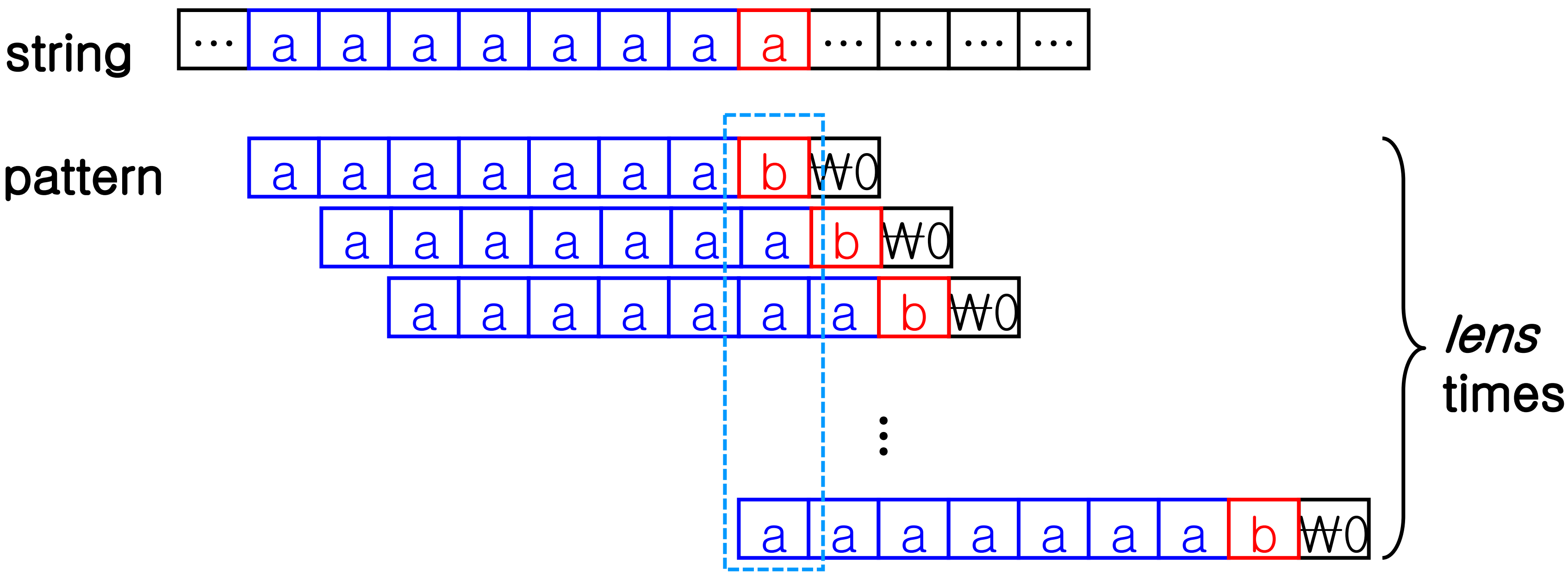


Simple Implementation

- Complexity of pattern_matching: O(lens * lenp)

  Ex) string = “aa…a”, pattern = “a…ab”

  Each ‘**a**’s in *string* is compared with most of ‘**a**’s in *pattern*



For an efficient algorithm, check the KMP (Knuth, Morris, Pratt) algorithm

questions or comments?

hchoi@handong.edu