INTRODUCTION TO PROGRAMING

Chapter 5

Array - String - Struct

Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
ĐHQG-HCM

GV: Thái Hùng Văn

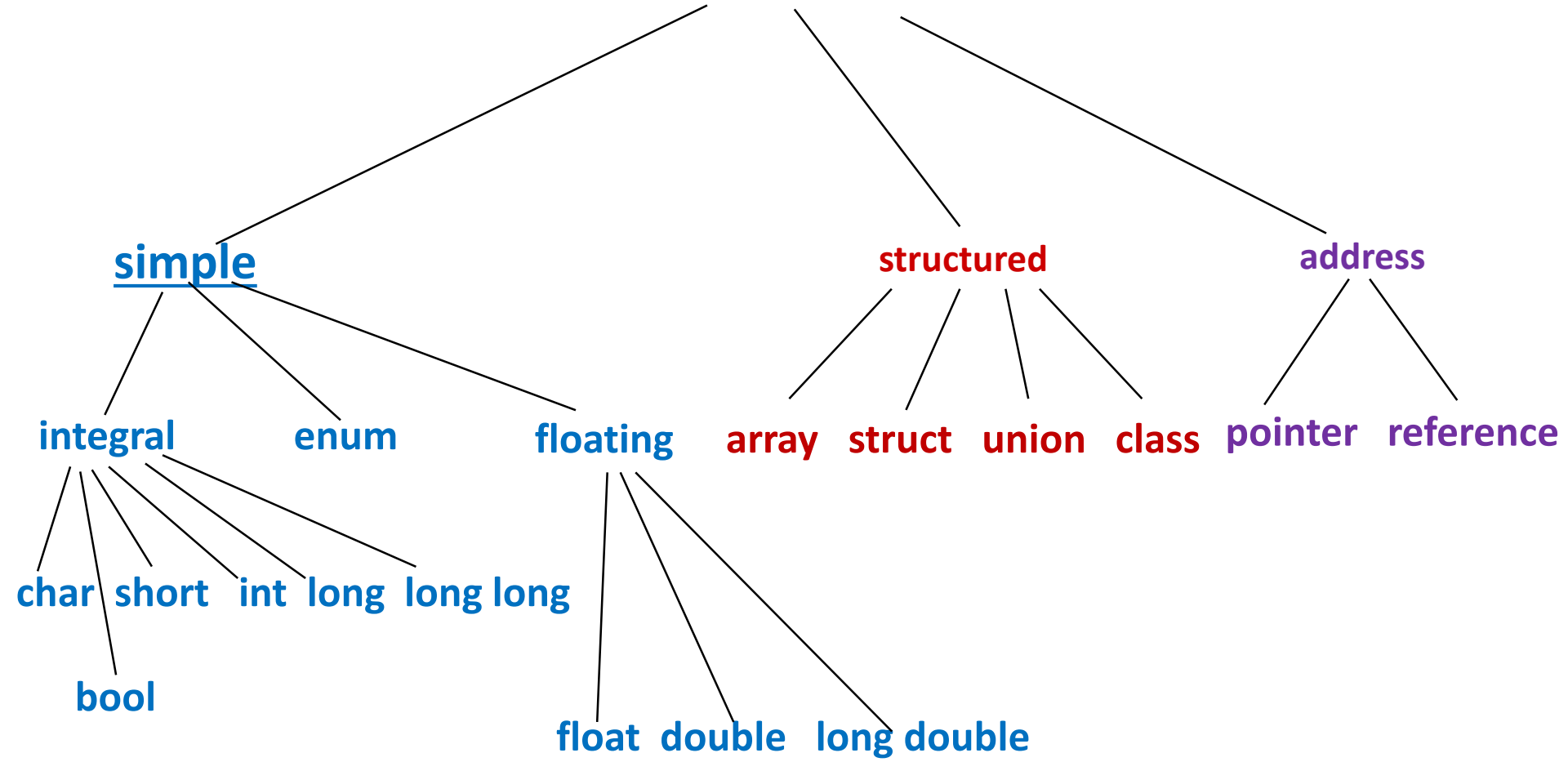
Objectives

In this chapter, you will:

- Learn about arrays
- Explore how to declare and manipulate data in 1D arrays *(Initializing, Reading, Writing, Inputing, Outputing,, Searching, Inserting, Deleting, Rearranging an element)*
- Learn how to pass an array as a parameter to a function
- Discover how to process 2D and nD arrays
- Learn about C-strings
- Examine the use of string functions to process C-strings
- Learn about struct
- Be able to use compound data structures in programs

Structured Data

The common data types in C ++



Simple data type

- Simple data type can store only one value at a time. It is also called primitive types - the most basic data types available within the C++ language.
- C++ simple data types are:
  - **integral** (char, short, int, long, long long, and bool)
  - **enum**
  - **floating** (float, double, long double)

Structured data type

- A structured data type is a type in which each value is a collection of component items.
- In a structured data type, the entire collection has a single identifier (name)
- Similarly, each component (element) can be accessed individually.
- C++ structured data types:
  - **array**
  - **struct**
  - **union**
  - **class**

Array

Need of Array

- Let's consider the situation where we need to get 10 student's phone numbers and store it for some thing.
- Since age is an integer type, we can store it something like:

```
int  phone1, phone2, phone3, phone4, phone5 ,
phone6, phone7, phone8, phone9, phone10;
```

- if we declare like above, it will be very difficult for us to manipulate the data.
- If more phonenumber of student joins, then it is very difficult to declare a lot of variables and keep track of it.
- To overcome this kind of situation, we should use **Array** data structure.

What is an Array?

- An array is a collection of values in same data type. *(e.g. a collection of int data values, or a collection of bool data values)*
- We refer to all stored values in an array by its **name**
- Arrays are always stored in a continuous memory location.
- If we would like to access a particular value stored in an array, we specify its index. The **1**st array index is always **0**
  - The **2**nd value is stored in index **1**
  - The **N**th value is stored in index **N-1**
- To store 10 students phone number, we can simply declare an array like below:

```
int  phone[10];
// To access 1st student's phone number, we can directly use index 0. i.e phone[0]
// To access Kth student's phone number, we can use index (K-1). i.e phone[K-1]
```

Types of Array

- In terms of dimensions, there are 2 types of arrays:
  - **One dimensional array** *// Typically used to represent a list of elements.*
  - **Multi dimensional array**
    - Two dimensional array *// can be used to represent a matrix*
    - Three dimensional array *// can be used to represent a point in 3Ds*
    - **N**th dimensional array
- In resizable ability, arrays can be classified into two kinds:
  - **Static array** *// must have a fixed size and be allocated on the stack*
  - **Dynamic array** *// may be allocated (on the heap) and then deallocated multiple times with different sizes (and these sizes can be larger than static array size)*
  - *C++ has a **Vector class** combines the advantages of both the static and dynamic arrays. It takes a non-const size parameter such as the dynamic array (but more efficient) and automatically deletes the used memory like the static array.*

How To Declare Arrays

- To declare an array, we should specify the following things:
  - The **data_type** of the values which will be stored in the array
  - The **array_name** of the array
  - The **dimensionality** of the array:
    - One dimension (i.e. list of values),
    - Two-dimension array (a matrix or a table), etc.
  - The **size** of **each dimension** MAX1, MAX2,.. *// must be constant*
- Syntax:

  data_type array_name [**MAX**]; *// for 1D array*
  data_type array_name [**MAX1**][**MAX2**]; *// for 2D array*
  data_type array_name [**MAX1**][**MAX2**]..[**MAXn**]; *// for **n**D array*
- Examples:
  - *int a [8];* // An integer array named **a** with size **8**
  - *bool gender [41];* // An array to store the gender for 41 students
  - *int scores[41] [5];* //2D array store scores of 5 exams for 41 students

1D Arrays

- The simplest form of an array is one-dimension (1D) array. The array itself is given name and its elements are referred to by their subscripts.
- We use 1D arrays to store and access list of data values in an easy way by giving these values a common name, e.g.

```
int a[4]; // all values are named a
a[0] = 10; // the 1st value is 10
a[2] = 20; // the 3rd value is 20
a[3] = 30; // the 4th value is 30
```

- A 1D array is a group of elements having the same datatype and same name. Individual elements are referred to using common name and unique index of the elements.

Accessing Array Components

- General syntax:

  array_name [**indexExp**]

  where **indexExp**, called an **index**, is any expression whose value is a non-negative integer.

- Index value specifies the position of component in array, it always starts at zero (0).
- [] is the **array subscripting operator**
- Examples:

```
int a [5], b;
cin >> a [0] >> b ;
a [1] = 2019;
a [b%2] = a[1] + b*a[0];
```

Array Index Out of Bounds

- All 1D array with N entries start at index 0 and ended at index N-1. It is a common error to try to access the N^{th} entry. *(since the index of the last array entry is **N-1**, not N)*
- If we have the statements:

```
double num[10];
int idx;
```

- The component `num[idx]` is valid if `idx` = 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9
- The index of an array is in bounds if the `index >=0` and the `index <= ARRAY_SIZE-1.` Otherwise, we say the `index` is **out of bounds.**
- In C/C++, there is no guard against indices that are out of bounds, and we must be careful**.**

Array Initialization During Declaration

- Array can be initialized during declaration in some forms:
  - The statement <`int a[5] = {0};`> declares `a` to be an array of `5` components and initializes all of them to zero.
  - The statement <`int a[5] = {2,0,19};`> declares `a` to be an array of `5` components, initializes `a[0]=2`, `a[1]=0`, `a[2]=19` and all other components are initialized to `0`
  - The statement <`int a[ ] = {2,0,19};`> declares `a` to be an array of 3 components, initializes `a[0]=2`, `a[1]=0`, `a[2]=19` .

Arrays as Parameters to Functions

- Arrays are passed by reference only, but the symbol **&** is ***not* used** when declaring an array as a formal parameter
- The size of the array is usually omitted *(If provided, it is ignored by the compiler)*
- If there are no changes to any of the components, we should add the keyword "**const**" first.
- The return value of the function cannot be an array.
- Examples:
  - void GetStudentIDList (**int Id [ ]** , **int &Num**)
  - void PutStudentIDList (**const int Id [ ]** , **int Num**)

  *// **Num** is number of components in array*

Base Address of an Array

- The base address of an array is the address, or memory location of the first array component
- If **`arr`** is a 1D array, its base address is the address of **`arr[0]`**. Similar, if **`arr`** is a **n**D array, its base address is the address of **`arr[0][0]..[0]`**.
- When we pass an array as a parameter, the base address of the actual array is passed to the formal parameter.
- Therefore, the two forms of parameter declaration for the function below are the same:
  - **<data_type> <array_name> [ ]**
  - **<data_type> * <array_name>**
- Examples: two parameter declarations below are the same:
  - void GetIDList (**int Id [ ]** , **int &Num**)
  - void GetIDList (**int * Id** , **int &Num**)

Processing 1D Arrays

Print Array's contents

- Use an index variable and a loop to browse all the elements.
- Example:

```
void printArray (const int arr [ ], int size)  {
    for  (int i=0; i<size; ++i)
            cout << arr [ i ] << " "; // print the ith element
}

int main()  {
    int a [ ] = { 3, 5, 6, 8, 1 };
    int n = sizeof (a) / sizeof (a[0]); // n = number of elements
    printArray ( a, n );              // print all of elements
    return 0;
}
```

Print Array using Recursion

Without Using Static Variable	Using Static Variable
void **printArr** (const int **arr** [ ], int **size**, int **i**) {	void **printArr**(int **arr** [ ], int **size**)
if (**i == size**) { *// base case*	{
i = 0;	**static int i**; *// using static variable*
cout << endl;	if (**i** == **size**) { *// base case*
return;	**i** = 0;
}	cout << endl;
cout << arr [i] << " ";	return;
i++;	}
// recursive call	cout << **arr** [**i**] << " ";
printArr (arr, size, i);	**i**++;
}	**printArr (arr, size);**
	}
call: **printArr (a, n, 0);**	*call:* **printArr (a, n);**

Input an Array

Known number of elements	Number of elements is unknown
bool **inputArr** (int **a** [ ], int **&n**) { cout<<"Number of elements: "; cin>>**n**; if (**n**<=0 || **n**>MAX) *// illegal* return false; for (int i=0; i<**n**; ++i) { cout<<"element["<<i<<"]="; cin>> **a [i]** ; } return true; }	void **inputArr** (int **a** [ ], int **&n**) { **n** = **0** ; do { cout<<"element["<<**n**<<"]="; cin>> **a [n]** ; **n**++ ; if (**n**==MAX) return; cout<<"input another?(y/n)"; char ans = getchar(); }while (ans=='Y' || ans=='y'); }
*call: result=***inputArr(arr, size);**	*call:* **inputArr (arr, size);**

Searching for A Key

in Unsorted Array - Using Sequential Search	in Sorted Array - Using Binary Search
*// This function will return the **index** of the **first value** that **equals** to the **key** if exists; otherwise, it returns **-1*** int **seqSearch** (int **arr**[ ], int **size**, int **key**) { for (int **i**=0; i<**size**; **i**++) if (**arr**[**i**] == **key**) *// or **Check (arr[i])** in general* return **i**; return -1; }	*// This function is similar to the left function* int **binSearch** (int **arr**[ ], int **size**, int **key**) { int **left=0, right=n-1**; do { int **mid = (left+right)/2**; if (**arr[mid] == key**) return **mid**; if (**x[mid]<key**) **right=mid-1;** else **left=mid+1;** } while (**left<=right**); return **-1**; }
*If element exists, the average number of repetitions is **N/2***	*The maximum number of repetitions is only **log2(N)***

Searching for the smallest value

in Unsorted Array - Using Sequential Search	in Sorted Array - Using Binary Search

```
// function return the index of the smallest value in the unsorted array
int minIndex (const int arr[ ], int size) {
    int minIdx = 0;
    int min = arr [0];
    for (int i=1; i<size; ++i)
        if (arr [ i ] < min) {
            min = arr [ i ] ;
            minIdx = i;
        }
    return minIdx;
}
//call: minPos=minIndex (a, n);
```

```
// function return the index of the smallest value in the unsorted array, start from start index (default value is zero)
int min_index (const int arr[ ], int size, int start=0)
{
   int minIdx = start;
   for (int i=start+1; i<size; ++i)
       if ( arr [ i ] < arr[minIdx] )
           minIdx = i;
   return minIdx;
}

// call:  minPos=minIndex (a, n);
// or:  minPos=minIndex (a, n, k);
```

Deleting element from an array

- To "delete" an element from the array you have to:
  - Find the position *(if not have)* of element.
  - Move all higher elements one place down and decrement the number of valid elements.
  - Reduce the size of array by one
- Example:

```
bool deleteElement (int arr [ ], int &size, int pos)
{
   if (pos<0 || pos>=size)
      return false;
   for (int i = pos; i < size - 1; i++)
      arr[ i ] = arr[ i+1 ];
   size -- ;
   return true;  //successful
}
```

```
int main()  {
  int a [ ] = { 3, 5, 6, 8, 1 } , key;
  int n = sizeof (a) / sizeof (a[0]);
  printArray ( a, n );
  cout<<"Enter the value u want to delete: ";
  cin>>key;
  if (deleteElement(a,n,seqSearch(a,n,key))
     printArray ( a, n );
  return 0;
}
```

Inserting an element to Array

- To "insert" an element into the array you have to:
  - Know the position and value of new element.
  - Move all higher elements one place up and increment the number of valid elements.

- ***For ex, with array:***

3	2	5	1	4	8	3	0	6
0	1	2	3	4	5	6	7	8

=> ***After inserting 9 to position 3:***

3	2	5	9	1	4	8	3	0	6
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

```
bool insertElement (int arr [ ], int &size, int pos, int key)
{
   if (pos<0 || pos>=MAX)
      return false;
   for (int i = size; i > pos; --i)
      arr [ i ] = arr [ i-1 ];
   arr [pos] = key;
   size ++ ;
   return true;  //successful
}
```

```
int main()  {
   int a [ ] = { 3, 2, 5, 1, 4, 8, 3, 0, 6 };
   int n = sizeof (a) / sizeof (a[0]);
   printArray ( a, n );
   int pos = 3, key = 9;
   if (insertElement (a, n, pos, key)) {
      cout<<"\Array after insert new element: ";
      printArray ( a, n );
   }
   return 0;
}
```

Sorting an Array

- There are many different sorting algorithms, with various pros and cons. Selection sort is one of the simplest algorithms.
- The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list.
- We can also do the opposite: finds the maximum element, swaps it with the last element.

```
void selectionSort (int a [ ], int n) {
   while (n--)
      swap( a[max_index(a,n)], a[n] );
}
int max_index  (const int a[ ], int n) {
  int maxIdx = 0;
  for (int i=1; i<n; ++i)
     if ( a[ i ] > a[maxIdx] ) maxIdx = i;
  return maxIdx;
}
```

```
int main()  {
  int a [ ] = { 1, 1, 2, 0, 1, 9 };
  int n = sizeof (a) / sizeof (a[1]);
  selectionSort ( a, n );
  printArray ( a, n );
}
void swap(int & a,int & b) {
    int temp=a; a = b; b = temp;
}
```

Processing 2D Arrays

General ways to process

- Each **row** and each **column** of a 2D array is a **1D array**.
- To process, use algorithms similar to processing 1D arrays
  - Process a particular row of the array, called row processing.
  - Process a particular column, called column processing.
- To handle the entire 2D array, two nested loops can be used
- In this section, the default declaration of the 2D array for example code is:

```
const int MAX_ROWS = 8; // this can be set to any number
const int MAX_COLS = 9; // this can be set to any number
int a [MAX_ROWS] [MAX_COLS];
int num_of_rows, num_of_cols; // actual number of rows & cols
// typedef  int  _2dArray [MAX_ROWS] [MAX_COLS];
// _2dArray  a;  //same as "int a[MAX_ROWS][MAX_COLS];"
```

Initialization During Declaration

- 2D arrays can be initialized when they are declared:
  - Elements of each row are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, if all components of a row aren't specified, unspecified ones are set to 0
- Examples:
  - *int matrix [3][5] = { {1,2,0,1,9}, {2,2,0,2,0}, {3,2,0,2,1} };*
  - enum MonHoc {NMLT, NMCNTT, TRR, KNM};

```
float diemSV[MAX_ROWS] [MAX_COLS]
diemSV[1] [NMLT] = 10;
diemSV[2] [NMCNTT] = 9.5;
// initialize colume number 2 (i.e., third colume - TRR) to value X
for (int i=0; i < TongSoSV; i++)
        diemSV[i][TRR] = X;
```

Print 2D Array's contents

Using 2 nested loops

```
void put2dArr(const int a[ ][MAX_COLS],
                        int nRows, int nCols)  {
   for  (int i=0; i< nRows; i++) {
      for  (int j=0; j< nCols; j++)
         // print the element at row I col J
         cout << a [ i ] [ j ] << " ";
      cout << endl ;
   }
}
// Note: When declaring a 2D array as  formal
parameter, can omit size of 1st dimension, but
not the 2nd
```

Using 1 loop, consider a 2D array as a 1D array where each element is a 1D array

```
void put1dArr (const int arr [ ], int size)
{
  for  (int i=0; i<size; i++)
     cout << arr [ i ] << " ";
  cout << endl ;
}
void put2dArr(const int a[ ][MAX_COLS],
                        int nRows, int nCols)
{
   for  (int i=0; i< nRows; i++)
      put1dArr ( a[ i ], nCols);
}
```

call:

```
put2dArr ( a, num_of_rows, num_of_cols);
```

Input 2D Array's contents

Using 2 nested loops

```
void get2dArr(int a[ ][MAX_COLS],
                        int nRows, int nCols)
{
   for  (int i=0; i< nRows; i++) {
      for  (int j=0; j< nCols; j++) {
         cout<<"a ["<< i <<"] ["<< j <<"]= ";
         cin>> a [ i ] [ j ] ;
      }
      cout << endl ;
   }
}
```

Using 1 loop

```
void get1dArr (int arr [ ], int size) {
  for  (int i=0; i<size; i++)
     cin >> arr [ i ] ;
}
void get2dArr(int a[ ][MAX_COLS],
                        int nRows, int nCols)  {
   for  (int i=0; i< nRows; i++) {
      cout<<"\nInput row " << i <<" : " ;
      get1dArr ( a[ i ], nCols);
   }
}
```

call:

get2dArr (a, num_of_rows, num_of_cols);

*// Note: **num_of_cols** or **nCols** must be less than or equals MAX_COLS*

Sum by Row /by Column

Sum by Row /by Colume

```
const int MAX_ROWS = 3;
const int MAX_COLS = 4;
typedef int  _2dArr[MAX_ROWS][MAX_COLS];
int sumRow (_2dArr a, int Row, int nCols)
{
   int sum = 0;
   for (int i=0; i<nCols; ++i)
      sum += a[Row] [ i ] ;
   return sum;
}
int sumCol (_2dArr a, int nRows, int Col)
{
   int sum = 0;
   for (int i=1; i<nRows; ++i)
      sum += a[ i ] [Col] ;
   return sum;
}
```

Sum by All

```
int sumAll (_2dArr a, int nRows, int nCols)
{
   int sum = 0;
   for  (int i=0; i< nRows; i++)
     // sum +=  sumRow (a, i, nCols);
     for  (int j=0; j< nCols; j++)
        sum += a[ i ] [ j ]
}

int main()
{
  _2dArr a ={{1,1,2,3}, {2,0,1,9}, {1,9,2,0}};
  for  (int r=0; r<MAX_ROWS; r++)
     cout<<"Sum of row "<<r<<" = "
        << sumRow (a, r, MAX_COLS) << endl;
}
```

Searching for the Largest Element in Row /Column

Largest Element in Row

```
//return index of the largest value in row
const int MAX_ROWS = 3;
const int MAX_COLS = 4;
typedef int  _2dArr[MAX_ROWS][MAX_COLS];
int maxIndex(_2dArr a, int Row, int nCols){
    int maxIdx = 0;
    int max = a[Row] [0];
    for (int i=1; i<nCols; ++i)
        if (a[Row] [ i ] > max) {
            max = a[Row] [ i ] ;
            maxIdx = i;
        }
    return maxIdx;
}
int main() {
  _2dArr a  = {{1,1,2,3}, {2,0,1,9}, {1,9,2,0}};
  cout <<"The largest value in row 2 is : "
          << a[2][maxIndex(a, 2, MAX_COLS)];
}
```

Largest Element in Colume

```
//return index of largest value in col
int maxIndex(_2dArr a, int nRows, int Col)
{
    int maxIdx = 0;
    int max = a[0] [Col];
    for (int i=1; i<nRows; ++i)
        if (a[ i ] [Col] > max) {
            max = a[ i ] [Col] ;
            maxIdx = i;
        }
    return maxIdx;
}

int main()
{
  _2dArr a ={{1,1,2,3}, {2,0,1,9}, {1,9,2,0}};
  cout <<"Largest value in colume 1 is : "
       << a[maxIndex(a, MAX_ROWS,1)][1];
}
```

C-String

Introdution

- Sequence of characters is called **string**. Strings are important in many programming contexts.
- In C, a string constant is enclosed in double quotes. Ex:
  - "2B|!2B"
  - "\nHello, World \n"

  Note: 'A' is not a string, it's just a character. And "A" is a string
- **C-string**s are null-terminated (`'\0'`) character arrays.
  - C-string "A" represents two characters: 'A' and '\0'
  - C-string "AB" has 3 byte: 65, 66, 0 *// 'A'=65, 'B'=66*
- **String** is an **array of char** and we can use it as a normal array. However, we should use **C-string** to get the rich support of C. In C++, <**string**> type is another type ad we will consider in the next subject.

C-string

- Consider the declaration statement:

```
char name[8];
```

- Since **C**strings are null terminated and **name** has 8 elements, the largest string that it can store has 7 characters
- If you store string "Thai" in **name**, the first 5 components of **name** are used and the last 3 are unused.
- Example:

```
char name[8]="Thai";
cout << name << endl; // outputs the string "Thai"
name[2]=0; // => name = "Th"
cout << name << endl;  // outputs the string "Th"
cout <<"string length = "<<strlen(name); // outputs: 2
```

Declaration and Initialization

- String can be initialized during declaration in some forms:
  - The statement < **`char s[9]="2B";`** > declares **`s`** to be a string of **`9`** components and a maximum of 8 characters, initializes s[0]='2', s[1]='B', s[2]=0 .
  - The statement < **`char s[]="2B|!2B";`** > declares **`s`** to be a string of **`7`** components , initializes s[0]=s[4]='2', s[1]=s[5]='B', s[2]='|', s[3]='!', s[6]=0.
  - The statement < **`char *s ="2B|!2B";`** > declares **`s`** the same form as above.
- Strings as Parameters to Functions: similar to the array, but it can be set to default parameters

```
void Introduce (char * s ="Hello") {
    cout<<s;
}
//call: Introduce();  or Introduce(" World");
```

Input String

- **cin>>str;** or **scanf("%s",str)** stores the next input C-string into **str** - the reading **stops at the blank** or newline character.
- To read strings with **blanks**, use **cin.getline /cin.get (str, n+1);** or **gets_s(str, n);** or **fgets(str, n, stdin);**
  - Stores the next **n** characters into **str** but the newline character is not stored in **str**
  - If the input string has fewer than **n** characters, the reading stops at the newline character.
  - **gets** function is replaced **by gets_s** from C11 (VS2015 - because not safe).
- If you do not want to specify the number of characters, you can use the function **scanf("%[^\n]%*c", str);** or **getline(cin, sstr)** with **sstr** is a string type.

Output String

- There are many easy ways to print a C-string, you can use **cout**, **printf**, **fprintf**, **puts**, **fputs** , ...
- Example:

```
const char str[] = "Hello World\n";
cout << str;
printf("%s", str);
puts(str);

for (int i=0; str[i]!=0; ++i)
    putchar(str[i]);
for (int i=0; str[i]!=0; ++i)
    putc(str[i], stdout);
for (int i=0; str[i]!=0; ++i)
    fputc(str[i], stdout);

fprintf(stdout, str);
fputs(str, stdout);
fwrite(str, 1, strlen(str), stdout);
```

Commonly Used String Functions

- **strlen()** : calculates the length of a given string *(doesn't count null character '\0')*
- **strcat() /strncat()** : append a copy of the source string to the end of destination string
- **strcmp() /stricmp() /strncmp()** : compare these two strings
- **strcpy() /strncpy()** : copy one string to another
- **strupr() /strlwr()** : convert all the characters in a string to *upper /lower case letters*
- **strchr()** : search for a particular **char** in the target **string**
- **strstr()** : finds the first occurrence of the sub-string

Struct

Introdution

- A **Structure** is a collection of related data items, possibly of different types. A structure type in C++ is called **struct**.
- A **struct** is **heterogeneous** in that it can be composed of data of different types.
- In contrast, **array** is **homogeneous** since it can contain only data of the same type.
- A struct is a derived data type composed of members that are each fundamental or derived data types.
- A single struct would store the data for one object. An array of structs would store the data for several objects.

Struct basics

- **Structures** hold data that belong **together**.
- Examples:
  - **Student record**: student id, name, phone, gender, start year, …
  - **Bank account**: account number, name, phone, currency, balance, …
- In database applications, structures are called **records**
- Each components of a struct are called **fields** (or **members**).
- Members can be of **different types** (simple, array or struct).
- A struct is named as a whole while individual members are named using field **identifiers**.
- Complex data structures can be formed by defining **arrays of structs.**

Struct declaration & typedef

- A ***struct type*** and ***struct variables*** can be defined in 2 ways:

```
struct <struct_Name> {
   <type1> <member1>;
   <type2> <member2>;
   ...
};
<struct_Name> <var1>, <var2>,.. ;
```

```
struct <struct_Name>
{
   <type1> <member1>;
   <type2> <member2>;
   ...
} <variable1>, <variable2>,.. ;
```

- Example:

```
struct StudentRecord{
      int Id;
      char Name[20];
      long PhoneNumber;
      bool Gender;
};
StudentRecord Sv1, Sv2; // Variables of StudentRecord type
```

Struct & typedef

- C provides a facility called ***typedef*** for creating synonyms for previously defined data type names.
- ***typedef*** can used in combination with ***struct*** to declare an alias (or a synonym) for a structure:

```
typedef struct
{
  <type1> <member1>;
  <type2> <member2>;
  ...
} <STRUCT_NAME>;
```

```
typedef struct <struct_Name>
{
  <type1> <member1>;
  <type2> <member2>;
  ...
}<STRUCT_NAME>;
```

- Example:

```
typedef  unsigned short  UINT16;
typedef  struct {
        UINT16 Id;
        char Name[20];
}STUDENT;
```

Accessing Struct Members

- Individual members of a ***struct*** variable may be accessed using the structure member operator (the dot, "."):

struct_variable.member

- Example:

```
typedef  unsigned short  UINT16;
typedef  struct {
     UINT16 Id;
     char Name[20];
}STUDENT;
STUDENT sd1, sd2, sd[MAX];
sd1.Id = 2019;
sd2.Id = sd1.Id + 1;
strcpy(sd1.Name, "Trump");
sd[1].Name[4] = 'b';
```

Sample Program #01

```
struct STUDENT {
      int Id;
      char Name[25];
} sd[40];

for (int i=0;i<N; i++) { // input student list
  cout<<"Id of student "<<i<<" : ";
  cin>> sd[i].Id;
  cout<<"Name of student "<<i<<" : ";
  cin.get(sd[i].Name, 40);
}
cout<<"Student list: ";
for (int i=0;i<N; i++) {// print student list
  cout<<"student "<<i<<" : Id=" << sd[i].Id;
  cout<<"Name=" << sd[i].Name <<endl;
}
```

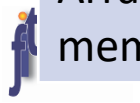
Sample Program #02

```
typedef  struct STUDENT {
      int Id;
      char Name[25];
};
STUDENT sd[40];

for (int i=0;i<N; i++) {
  cout<<"Id of student "<<i<<" : ";
  cin>> sd[i].Id;
  cout<<"Name of student "<<i<<" : ";
  cin.get(sd[i].Name, 40);
}
cout<<"Student list: ";
for (int i=0;i<N; i++) {
  cout<<"student "<<i<<" : Id=" << sd[i].Id;
  cout<<"Name=" << sd[i].Name <<endl;
}
```

Difference between Structure and Array

ARRAY	STRUCTURE
Array refers to a collection consisting of elements of homogenous data type.	Structure refers to a collection consisting of elements of heterogenous data type.
Array uses “[ ]” for elements access, by their index number using subscripts.	Structure uses “.” for elements access, by their names using dot operator.
Array is pointer (points to the 1st element)	Structure is not a pointer
Instantiation of Array objects is not possible.	Instantiation of Structure objects is possible.
Array size is fixed and is basically the number of elements multiplied by the element size.	Structure size is not fixed as each element of Structure can be of different type and size.
Bit filed is not possible in an Array.	Bit filed is possible in an Structure.
Array declaration is done simply using [] and not any keyword.	Structure declaration is done with the help of “struct” keyword.
Arrays is a primitive datatype	Structure is a user-defined datatype.
Array traversal and searching is easy and fast.	complex and slow.
data_type array_name[size];	struct structName {type1 ele1; type2 ele2;..}
Array elements are stored in continuous memory locations.	Structure elements may or may not be stored in a continuos memory location.



Exercises

Write a function to handle each of the issues below:

1. Inserting a value into the ascending array. *(function header should be like this:* ***int insertAccending (int arr [ ], int size, int key)*** *;Write a C++ program to test your function)*
2. The sum of the values stored in an array
3. The mean and the standard deviation for the array values
4. The range of the values stored in an array
5. Multiplying array values by a constant
6. Dot-product of two arrays
7. The intersection of two arrays (acting as sets)
8. The union of two arrays (acting as sets)
9. The difference between two arrays acting as sets
10. Binary representation for a given integer number
11. Generate Fibonacci series
12. Generate geometric series


End!