University of Baghdad- College of science-
Computer science department

Advanced Programming Fundamentals

Dr. Nasreen Jawad Kadhim

Second term- First year class
2020-2021

Lec4:

5-string & C-string (Character Array):

5.1 string:

- ✓ A string is a sequence of zero or more characters.
- ✓ The string data type is a programmer-defined data type. It is **Not** directly available for use in a program like the simple data types discussed earlier.

> **Tip:**
> The Standard C++ library did not provide a string data type. Compiler vendors often supplied their own programmer-defined string type, and the syntax and semantics of string operations often varied from vendor to vendor.

- ✓ To use the string data type, you need to access its definition from the header file **string**. Therefore, to use the string data type in a program, you must include the following preprocessor directive: **# include <string>**
- ✓ Strings in C++ are enclosed in **double quotation marks ("")**.
- ✓ A string containing **no characters** is called a null or empty string. The following are examples of strings. Note that "" is the empty string.

  "William Jacob"
  "Mickey"
  ""

- ✓ Every character in a string has a relative **position** in the string. The **position** of the first character is 0, the position of the second character is 1, and so on.
- ✓ The **length** of a string is the number of *valid or relevant* characters in it.

String	Position of a Character in the String	Length of the String
`"William Jacob"`	Position of `'W'` is 0. Position of the first `'i'` is 1. Position of `' '` (the space) is 7. Position of `'J'` is 8. Position of `'b'` is 12.	13
`"Mickey"`	Position of `'M'` is 0. Position of `'i'` is 1. Position of `'c'` is 2. Position of `'k'` is 3. Position of `'e'` is 4. Position of `'y'` is 5.	6

- ✓ When determining the **length** of a string, you should also count all the spaces, if found, in the string. For example, the length of the following string is **22**.

  "It is a beautiful day."

✓ The string data type is very powerful and more complex than simple data types. It provides many **operations** (functions) to manipulate strings. For example, it provides operations to find the length of a string, extract part of a string, and compare strings , etc (see 5.1.2).

✓ There are several **operators** that can be applied to the string data type such as:

- ❖ the operator = (assignment), operator + (to allow string concatenation operation),
- ❖ and the array index (subscript) operator [ ] (to access an individual character within a string together with its related position).

Let's see how these operators work on the string data type.

Suppose you have the following declarations:

```
string str1, str2, str3;            // To declare string variable(s)
```

- str1 = "Hello There"; // assignment statement, stores the string "Hello There" in str1
- str2 = str1; // copies the value of str1 into str2.
- str1 = "Sunny"; // stores the string "Sunny Day" into str1
  str2 = str1 + " Day"; // stores the string generated from concat. ("Sunny Day") into str2
- str1 = "Hello"
  str2 = "There"
  str3 = str1 + " " + str2; // stores "Hello There" into str3
  This statement is equivalent to the statement:
  str3 = str1 + ' ' + str2;
- str1 = str1 + " Ahmed"; // updates the value of str1 by appending the string " Ahmed"
  Therefore, the new value of str1 is "Hello Ahmed".
- str1 = "Hello there";
  str1[6] = 'T'; // replaces the character t with the character T.
  Recall that the position of the first character in a string variable is 0. Therefore, because t is the **seventh** character in str1, its position is 6.

Note:
For the operator + to work with the string data type, one of the operands of + must be a **string variable**. For example, the following statements will not work:

```
str1 = "Hello " + "there!";      //illegal
str2 = "Sunny Day" + '!';        //illegal
```

✓ Also, the **relational operators** can be applied to variables of **string data type**. Variables of data type **string** are compared character-by-character, starting with the first character and using the ASCII collating sequence.

The character-by-character comparison continues until either a mismatch is found or the last characters have been compared and are equal.

The following example shows how variables of data type string are compared. Suppose that you have the following statements:

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str5 = "Big";
```

The table below illustrates the evaluation of the following string relational expressions and the result of the comparison between two strings:

Expression	Value /Explanation
`str1 < str2`	`true` `str1 = "Hello"` and `str2 = "Hi"`. The first characters of `str1` and `str2` are the same, but the second character `'e'` of `str1` is less than the second character `'i'` of `str2`. Therefore, `str1 < str2` is `true`.
`str1 > "Hen"`	`false` `str1 = "Hello"`. The first two characters of `str1` and `"Hen"` are the same, but the third character `'l'` of `str1` is less than the third character `'n'` of `"Hen"`. Therefore, `str1 > "Hen"` is `false`.
`str3 < "An"`	`true` `str3 = "Air"`. The first characters of `str3` and `"An"` are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An"`. Therefore, `str3 < "An"` is `true`.
`str1 == "hello"`	`false` `str1 = "Hello"`. The first character `'H'` of `str1` is less than the first character `'h'` of `"hello"` because the ASCII value of `'H'` is 72, and the ASCII value of `'h'` is 104. Therefore, `str1 == "hello"` is `false`.
`str3 <= str4`	`true` `str3 = "Air"` and `str4 = "Bill"`. The first character `'A'` of `str3` is less than the first character `'B'` of `str4`. Therefore, `str3 <= str4` is `true`.
`str2 > str4`	`true` `str2 = "Hi"` and `str4 = "Bill"`. The first character `'H'` of `str2` is greater than the first character `'B'` of `str4`. Therefore, `str2 > str4` is `true`.

If two strings of different lengths are compared and the character-by-character comparison is equal until it reaches the last character of the shorter string, the shorter string is evaluated as less than the larger string, as shown in the following:

Expression	Value/Explanation
`str4 >= "Billy"`	`false` `str4 = "Bill"`. It has four characters, and `"Billy"` has five characters. Therefore, `str4` is the shorter string. All four characters of `str4` are the same as the corresponding first four characters of `"Billy"`, and `"Billy"` is the larger string. Therefore, `str4 >= "Billy"` is `false`.
`str5 <= "Bigger"`	`true` `str5 = "Big"`. It has three characters, and `"Bigger"` has six characters. Therefore, `str5` is the shorter string. All three characters of `str5` are the same as the corresponding first three characters of `"Bigger"`, and `"Bigger"` is the larger string. Therefore, `str5 <= "Bigger"` is `true`.

Lec 9 :

5.1.1 Input/Output with string data type:

✓ You can use an **input stream variable**, such as **cin**, and the **extraction operator** >> to read a string into a variable of string data type.

For example, if the input is the string "Ahmed", the following code stores this input into the string variable **studentName**:

```
string studentName;
cin >> studentName;
```

✓ Recall that the extraction operator skips any leading whitespace characters and that reading **stops at a whitespace character**. As a consequence, you cannot use the extraction operator to read strings that contain **blanks**.

For example, suppose that the variable **studentName** is defined as illustrated above. If the input is the string "Ahmed Sami" then after the statement:

```
cin >>  studentName;
```

executes, the value of the variable **studentName** is "Ahmed ".

✓ To read a string containing **blanks**, you can use the function **getline**.

The syntax to use the function **getline** is:

```
getline(istreamVar, strVar);
```

where; **istreamVar** is an **input stream variable** and **strVar** is a **string variable**.

The reading is delimited by the **newline character** **'\n'**.

The function **getline** reads until it reaches the end of the current line. The newline character is also read but not stored in the string variable.

✓ Consider the following statement:

```
string myString;
```

If the input is **29** characters: bbbbHello there. How are you?

where b represents a blank, after the statement:

```
getline (cin, myString);
```

Now, the value of myString is:

myString = "   Hello there. How are you?"

All **29** characters, including the first **four** blanks, are stored into myString.

✓ Similarly, you can use **output stream variable**, **cout**, and the insertion operator << to output the contents of a **string variable**.

```
cout << "You entered the string : " << myString;
```

The following program shows the effect of the preceding statements in the previous sections.

Example 1:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{

string name = "William Jacob";
string str1, str2, str3, str4;

cout << "Name = " << name << endl;
str1 = "Hello There";
cout << "str1 = " << str1 << endl;

str2 = str1;
cout << "str2 = " << str2 << endl;

str1 = "Sunny";
str2 = str1 + " Day";
cout << "str2 = " << str2 << endl;

str1 = "Hello";
str2 = "There";
str3 = str1 + " " + str2;       //  str3 = str1 + ' ' + str2;
cout << "str3 = " << str3 << endl;

str1 = str1 + " Ahmed";
cout << "str1 = " << str1 << endl;

str1 = "Hello there";
cout << "str1[6 ] = " << str1[6 ] << endl;
str1[6] = 'T';
cout << "str1 = " << str1 << endl;

//String input operations:
string studentName;

cout << "Enter your name :  ";
cout<<"\n==============\n";

cin >> studentName;
getline(cin, studentName);   // stores string with blanks
cout<<"The name you entered is : " << studentName << endl;

system ("pause");
return 0;
}
```

Lec 3 :

5.1.2 Additional string Operations:

✓ The data type string contains several (***predefined***) functions for string manipulation. The following table describes some of these functions. In this table, we assume that **strVar** is a string variable and **str** is a string variable, a string constant or a character array.

Expression	Effect
`strVar.at(index)`	Returns the element at the position specified by `index`.
`strVar[index]`	Returns the element at the position specified by `index`.
`strVar.append(n, ch)`	Appends `n` copies of `ch` to `strVar`, in which `ch` is a `char` variable or a `char` constant.
`strVar.append(str)`	Appends `str` to `strVar`.
`strVar.clear()`	Deletes all the characters in `strVar`.
`strVar.compare(str)`	Compares `strVar` and `str`.
`strVar.empty()`	Returns `true` if `strVar` is empty; otherwise, it returns `false`.
`strVar.erase()`	Deletes all the characters in `strVar`.
`strVar.erase(pos, n)`	Deletes `n` characters from `strVar` starting at position `pos`.
`strVar.erase(pos);`	Removes all of the characters from `strVar` starting at index `pos`.
`strVar.find(str)`	Returns the index of the first occurrence of `str` in `strVar`.
`strVar.find(str, pos)`	Returns the index of the first occurrence at or after `pos` where `str` is found in `strVar`.
`strVar.insert(pos, n, ch);`	Inserts `n` occurrences of the character `ch` at index `pos` into `strVar`; `ch` is a character.
`strVar.insert(pos, str);`	Inserts all the characters of `str` at index `pos` into `strVar`.
`strVar.length()`	Returns a value giving the number of characters `strVar`.
`strVar.replace(pos, n, str);`	Starting at index `pos`, replaces the next `n` characters of `strVar` with all the characters of `str`. If `n` > length of `strVar`, then all the characters until the end of `strVar` are replaced.
`strVar.substr(pos, len)`	Returns a string that is a substring of `strVar` starting at `pos`. The length of the substring is at most `len` characters. If `len` is too large, it means "to the end" of the string in `strVar`.
`strVar.size()`	Returns a value giving the number of characters `strVar`.
`strVar.swap(str1);`	Swaps the contents of `strVar` and `str1`. `str1` is a `string` variable.

✓ The next examples illustrate the effect of applying some of these functions:

(clear, empty, erase, length, and size functions)

Example 2:

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string nullString = "";
string firstName = "Ahmed";
string fullName = firstName + " Sami Ali";
string str1 = "It is sunny.";
string str2 = "";
string str3 = "Computer science";
string str4 = "C++ programming.";
string str5 = firstName + " is taking " + str4;

cout << "str3: " << str3 << endl;
str3.clear( );
cout <<"After clear, the str3 is : " << str3 << endl;

cout <<str1<< "\n str1.empty( ), returns : "<< str1.empty( ) << endl;
cout <<str2<< "\n str2.empty( ), returns : "<<str2.empty( ) << endl;

cout << "str4: " << str4 << endl;
str4.erase( );
cout << "After erase( ), str4: " << str4 << endl;

str4 = "C++ programming.";
cout << "str4: " << str4 << endl;
str4.erase(11, 4);
cout << "After erase(11, 4), str4: " << str4 << endl;

str4 = "C++ programming.";
cout << "str4: " << str4 << endl;
str4.erase(11);
cout << "After erase(11), str4: " << str4 << endl;

cout<<"Length of \"" << nullString <<"\" = " << (nullString.length( ))<<endl ;
cout<<"Size of \"" << nullString <<"\" = " << (nullString.size( ))<<endl ;
cout<< "Length of \"" << firstName << "\" = " << (firstName.length( )) << endl;
cout<< "Length of \"" << fullName << "\" = " << (fullName.length( )) << endl;
cout<< "Size of \"" << fullName << "\" = " << (fullName.size( )) << endl;
cout<< "Length of \"" << str1 << "\" = " <<  (str1.length( )) << endl;
cout<< "Size of \"" << str5 << "\" = " <<  (str5.size( )) << endl;

len = fullName.length( );
cout << "len = " << len << endl;
system ("pause");
return 0;
}
```

Lec 4:

✓ Suppose str1 and str2 are of type **string**. The following are valid calls to the function **find**:

```
str1.find(str2)
str1.find("the")
str1.find('a')
str1.find(str2 + "xyz")
str1.find(str2 + 'b')
```

Consider the following statements:

```
string sentence = "Outside it is cloudy and warm.";
string str = "cloudy";
int position;
```

Next, the effect of the **find** function is illustrated.

Statement	Effect
`cout << sentence.find("is") << endl;`	Outputs 11
`cout << sentence.find('s') << endl;`	Outputs 3
`cout << sentence.find(str) << endl;`	Outputs 14
`cout << sentence.find("the") << endl;`	
`cout << sentence.find('i', 6) << endl;`	Outputs 8
`position = sentence.find("warm");`	Assigns 25 to position

Note that the search is case sensitive. Therefore, the position of o (lowercase o) in the string sentence is 16. The following program illustrates the effect of **find** function.

(find function)

Example 3:

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string sentence = "Outside it is cloudy and warm.";
string str = "cloudy";
int position;

cout << "sentence = \"" << sentence << "\"" << endl;
cout << "The position of \"is\" in sentence = " << sentence.find("is") << endl;
cout << "The position of 's' in sentence = " << sentence.find('s') << endl;
cout << "The position of \"" << str << "\" in sentence = " << sentence.find(str) << endl;
cout << "The position of \"the\" in sentence = " <<(int)sentence.find("the") << endl;
cout << "The first occurrence of \'i\' in sentence after position 6 = " <<sentence.find('i', 6)
<< endl;
position = sentence.find( "warm" );
cout << "Position = " << position << endl;
system ( "pause" );
return 0;
}
```

✓ Suppose that you execute the following:

```
string firstString = "Cloudy and warm.";
string secondString = "Hello there";
string thirdString = "Henry is taking programming I.";
string str1 = " very ";
string str2 = "Lisa";
```

Next, we show the effect of **insert** and **replace** functions.

Statement	Effect
`firstString.insert(10, str1);`	`firstString = "Cloudy and very warm."`
`secondString.insert(11, 5, '!');`	`secondString = "Hello there!!!!!"`
`thirdString.replace(0, 5, str2);`	`thirdString = "Lisa is taking programming I."`

The program below shows exec. of prev. statements.

(insert and replace functions)

Example 4:

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string firstString = "Cloudy and warm.";
string secondString = "Hello there";
string thirdString = "Henry is taking programming I.";
string str1 = " very ";
string str2 = "Lisa";

cout << "firstString = " << firstString << endl;

firstString.insert(10, str1);
cout << "After insert; firstString = " << firstString << endl;

cout << "secondString = " << secondString << endl;

secondString.insert(11, 5, '!');
cout << "After insert; secondString = " << secondString << endl;

cout << "thirdString = " << thirdString << endl;

thirdString.replace(0, 5, str2);
cout << "After replace, thirdString = " << thirdString << endl;

system ("pause");
return 0;
}
```

Lec 5 :

✓ Consider the following statements:

```
string sentence;

string str;

sentence = "It is cloudy and warm.";
```

Next, we show the effect of the **substr** function.

Statement	Effect
`cout << sentence.substr(0, 5) << endl;`	Outputs: It is
`cout << sentence.substr(6, 6) << endl;`	Outputs: cloudy
`cout << sentence.substr(6, 16) << endl;`	Outputs: cloudy and warm.
`cout << sentence.substr(17, 10) << endl;`	Outputs: warm.
`cout << sentence.substr(3, 6) << endl;`	Outputs: is clo
`str = sentence.substr(0, 8);`	str = "It is cl"
`str = sentence.substr(2, 10);`	str = " is cloudy"

The following program illustrates applying the string function **substr**.

(substr function)

Example 5:

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string str;
string sentence = "It is cloudy and warm.";

cout << "substr(0, 5) in \"" << sentence << "\" = \"" << sentence.substr(0, 5) << "\"" << endl;

cout << "substr(6, 6) in \"" << sentence << "\" = \"" << sentence.substr(6, 6) << "\"" << endl;

cout << "substr(6, 16) in \"" << sentence << "\" = " << endl;

cout << " \"" << sentence.substr(6, 16) << "\"" << endl;

cout << "substr(17, 10) in \"" << sentence << "\" = \"" << sentence.substr(17, 10) << "\"" << endl;

cout << "substr(3, 6) in \"" << sentence << "\" = \"" << sentence.substr(3, 6) << "\"" << endl;

str = sentence.substr(0, 8);

cout << "str = \"" << str << "\"" << endl;

str = sentence.substr(2, 10);

cout << "str = \"" << str << "\"" << endl;

system ("pause");
return 0;
}
```

✓ The **swap** function is used to swap—that is, interchange—the contents of two string variables. Suppose the following statements are executed:

string str1 = "Warm";

string str2 = "Cold";

After the following statement executes, the value of str1 will be "Cold" and the value of str2 will be "Warm".

str1.swap(str2);

The following program illustrates exec. of the previous statements.

(swap function)

Example 6:

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string str1 = "Warm";
string str2 = "Cold";

cout << "str1 :" << str1<< endl;
cout << "str2 :" << str2<< endl;

str1.swap(str2);
cout << "After swap, str1 = " << str1 << endl;
cout << "After swap, str2 = " << str2 << endl;
system ("pause");
return 0;
}
```

Notes:

- ✓ String is a sequence of zero or more characters.
- ✓ Strings in C++ are enclosed in double quotation marks.
- ✓ A string containing no characters is called a null or empty string.
- ✓ Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on.
- ✓ The length of a string is the number of *valid or relevant* characters in it (number of characters between "").

L 6:

5.2 C-string (Character Array):

- ✓ **Character array**: An ordinary **array** whose components are of type **char**.
- ✓ The first character in the ASCII character set is the **null character**, which is nonprintable. Also, recall that in C++, the **null character** is represented as '**\0**', a **backslash** followed by a **zero**. It is less than any other character in the **char** data set.
- ✓ A **C-string** is an array of characters terminated by the **null character** '**\0**' but array of character is not a C-string. The **null character** plays an important role in processing **C- string**.
- ✓ **C-strings** are stored in (**one-dimensional**) character arrays. So, you can declare and initialize it just like **array** of chararacter. **For example**:

```
char greeting [ ] = { 'H','e','l','l','o',' ','W','o','r','l','d','\0' };
```

C++ enables you to use a shorthand form of the above line of code. It is:

```
char greeting [ ]= "Hello World" ;
```

Therefore, **C-strings** can be enclosed in double quotation marks.

- ✓ Consider the following statement:

```
char name[16];
```

This statement declares an **array** name of 16 components of type **char**. Because **C-strings** are null-terminated and name has 16 components, the largest string that can be stored in name is of length 15. If you store a **C-string** of length 10 in name, the first 11 components of name are used and the last 5 are left unused.

- ✓ The statement:

```
char name[16] = { 'S', 'a', 'm', 'i', '\0' };
```

declares an **array** name containing 16 components of type **char** and stores the **C-string** "Sami" in it. During **declaration**, C++ allows the **C-string** notation to be used in the initialization statement. The above statement is, therefore, equivalent to:

```
char name[16] = "Sami";          //Line A
```

Recall that the size of an array can be omitted if the array is initialized during the declaration. The statement:

```
char name[ ] = "Sami";          //Line B
```

declares a **C-string** variable **name** of a length large enough—in this case, 5—and stores "Sami" in it.

There is a **difference** between the last two statements (Line A & Line B): Both statements store "Sami" in **name**, but the **size** of name in the statement in Line A is **16**, and the **size** of name in the statement in Line B is **5**.

✓ Most **rules** that applied to other arrays also applied to character arrays.

Consider the following statement: **char** studentName[20];

Suppose you want to store "Ahmed Sami" in studentName. Because aggregate operations, such as **assignment** and **comparison**, are not allowed on arrays, the following statement is illegal:

```
studentName = "Ahmed Sami";    //illegal
```

5.2.1 Input/Output with C-string:

✓ The aggregate operations are allowed for **C-string input/output**.

✓ **C-string Input**: the following declaration will be used for our discussion:

```
char name[31];
```

As mentioned above, because aggregate operations are allowed for **C-string input**, the statement:

```
cin >> name ;
```

stores the next input **C-string** into name. The length of the input **C-string** must be less than or equal to 30 (at most, 30 characters).

- If the **length of the input C-string** is 4, the computer stores the four characters that are input and the null character '\0'.
- If the **length of the input C-string** is more than 30, then because there is no check on the array index bounds, the computer continues storing the string in whatever memory cells follow name. This process can cause serious problems, because data in the adjacent memory cells will be corrupted.

Tip:

Recall that the **extraction** operator, >>, skips all leading whitespace characters and stops reading data into the current variable as soon as it finds the first whitespace character or invalid data.

As a result, **C-strings** that contain **blanks** cannot be read using the **extraction** operator, >>. For example, if a **first name** and **last name** are separated by **blanks**, they cannot be read into name.

Question: How do you input **C-strings** with blanks into a character array?

Solution : To read and store a line of input, including whitespace characters, you can use the stream function **getline**.

So, suppose that you have the following declaration:

```
char studentName [40];
```

The following statement will read and store the next 39 characters, or until the newline character, into studentName. The null character will be automatically appended as the last character of studentName.

```
cin.getline(studentName, 40); // stores 39 characters
```

✓ **C-string Output**: Again, the following declaration will be used for the discussion:

```
char name[31];
```

You can output **C-strings** by using an output stream variable **cout**, together with the **insertion** operator, <<. For example, the statement:

```
cout << name;
```

outputs the contents of name on the screen. The insertion operator, <<, continues to write the contents of name until it finds the **null character**.

- If the **length of name** is 4, the above statement outputs only four characters.
- If name does not contain the null character, then you will see strange output because the insertion operator continues to output data from memory adjacent to name until '**\0**' is found.

L7 :

5.2.2 Additional C-string Operations:

✓ C++ provides a set of (***predefined***) functions that can be used for **C-string** manipulation. Header file <cstring> contains these functions. The table below summarizes these functions.

Function Name and Parameters	Parameter(s) Type	Function Return Value
`strcpy(destStr, srcStr)`	`destStr` and `srcStr` are null-terminated `char` arrays	The base address of `destStr` is returned; `srcStr` is copied into `destStr`
`strlen(str)`	`str` is a null-terminated `char` array	An integer value ≥ 0 specifying the length of the `str` (excluding the `'\0'`) is returned
`strcat(destStr, srcStr)`	`destStr` and `srcStr` are null-terminated `char` arrays; `destStr` must be large enough to hold the result	The base address of `destStr` is returned; `srcStr`, including the null character, is concatenated to the end of `destStr`
`strcmp(str1, str2)`	`str1` and `str2` are null-terminated `char` arrays	The returned value is as follows: • An `int` value < 0 if `str1 < str2` • An `int` value 0 if `str1 = str2` • An `int` value > 0 if `str1 > str2`

✓ In C++, **C-strings** are compared **character-by-character** using the system's collating sequence. Let us assume that you use the **ASCII** character set.

1. The **C-string** "Air" is less than the **C-string** "Boat" because the first character of "Air" is less than the first character of "Boat".
2. The **C-string** "Air" is less than the **C-string** "An" because the first characters of both strings are the same, but the second character 'i' of "Air" is less than the second character 'n' of "An".
3. The **C-string** "Bill" is less than the **C-string** "Billy" because the first four characters of "Bill" and "Billy" are the same, but the fifth character of "Bill", which is '\0' (the null character), is less than the fifth character of "Billy", which is 'y'. (Recall that **C-strings** in C++ are null terminated.)
4. The **C-string** "Hello" is less than "hello" because the first character 'H' of the **C-string** "Hello" is less than the first character 'h' of the **C-string** "hello".

✓ Suppose you have the following statements:

```
char studentName[21];
char myname[16];
char yourname[16];
```

The following statements show how string functions work:

Statement	Effect
`strcpy(myname, "John Robinson");`	`myname = "John Robinson"`
`strlen("John Robinson");`	Returns 13, the length of the string `"John Robinson"`
`int len;` `len = strlen("Sunny Day");`	Stores 9 into `len`
`strcpy(yourname, "Lisa Miller");` `strcpy(studentName, yourname);`	`yourname = "Lisa Miller"` `studentName = "Lisa Miller"`
`strcmp("Bill", "Lisa");`	Returns a value < 0

`strcpy(yourname, "Kathy Brown");`
`strcpy(myname, "Mark G. Clark");`
`strcmp(myname, yourname);`
	`yourname = "Kathy Brown"`
	`myname = "Mark G. Clark"`
	Returns a value > 0

L 6 :

5.2.3 Character handling functions:

There are several built-in (predefined) functions that can help you to manipulate individual characters. You should include the header file **cctype** in order to use these functions. There are **two** sets of functions:

1- Character classification functions:

They check whether the character passed as parameter belongs to a certain category:

Function Return Value	Parameter(s) Types	Function Name and Parameters
Function returns an int value as follows: • If ch is an alphabetic letter or a digit character, that is ('A'-'Z', 'a'-'z', '0'-'9'), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isalnum(ch)**
Function returns an int value as follows: • If ch is an alphabetic letter, that is ('A'-'Z', 'a'-'z'), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isalpha(ch)**

Function returns an int value as follows: • If ch is a digit ('0'-'9'), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isdigit(ch)**
Function returns an int value as follows: • If ch is an uppercase letter ('A'-'Z'), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isupper(ch)**
Function returns an int value as follows: • If ch is a lowercase letter ('a'-'z'), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**islower(ch)**
Function returns an int value as follows: • If ch is a printable character, including blank (**in ASCII, ' ' through '~'**) (**i.e. [32 - 126]**), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isprint(ch)**
Function returns an int value as follows: • If ch is a whitespace character (**in ASCII, a character value 9-13 and space 32**), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**isspace(ch)**
Function returns an int value as follows: • If ch is a control character (**in ASCII, a character value 0-31 or DEL 127**), it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**iscntrl(ch)**
Function returns an int value as follows: • If ch is a punctuation character (**in ASCII, [! , /], [: , @], [[, `], and [{ , ~]) (i.e. [33, 47], [58, 64], [91, 96], and [123, 126])** , it returns a nonzero value (true) • 0 (false), otherwise	**ch is a char value**	**ispunct(ch)**

2- Character conversion functions:

Two functions that convert between letter cases:

Function Return Value	Parameter(s) Types	Function Name and Parameters
If ch is a lowercase alphabetic letter then toupper returns the corresponding uppercase letter, otherwise ch is returned unchanged.	**ch is a char value**	**toupper(ch)**
If ch is an uppercase alphabetic letter then tolower returns the corresponding lowercase letter, otherwise ch is returned unchanged.	**ch is a char value**	**tolower(ch)**

L9.

5.2.4 Arrays of Strings:

✓ Suppose that you need to perform an operation, such as alphabetizing a list of names. Because every name is a string, a convenient way to store the list of names is to use an array. Strings in C++ can be manipulated using either the data type string or character arrays (**C-strings**). This section illustrates both **ways** to manipulate a list of strings.

1- Arrays of Strings (String):

Processing a list of strings using the data type string is straightforward. Suppose that the list consists of a maximum of 100 names. You can declare an array of 100 components of type string as follows:

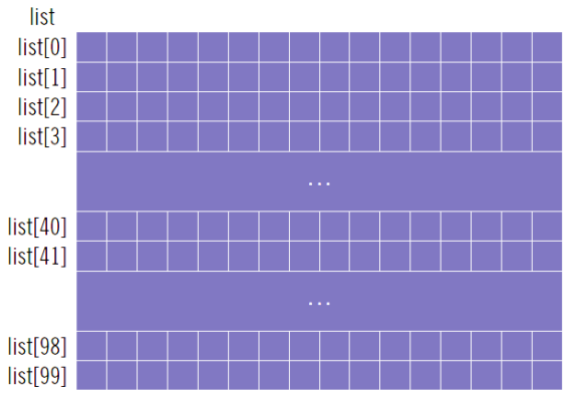
string list [100] ;

Basic operations, such as **assignment**, **comparison**, and **input/output**, can be performed on values of the string type. Therefore, the data in list can be processed just like any **one-dimensional array**.

2- Arrays of Strings (C-String):

Suppose that the largest string (for example, name) in your list is 15 characters long and your list has 100 strings. You can declare a **two-dimensional array** of characters of 100 rows and 16 columns as follows (see Figure 4-1):

char list [100] [16] ;



Figure(4-1): Array list of strings

Now list [j] for each j, **0 <= j <= 99**, is a string of at most 15 characters in length.

The following statement stores "Snow White" in list[1] (see Figure 4-2):

strcpy (list[1], "Snow White");

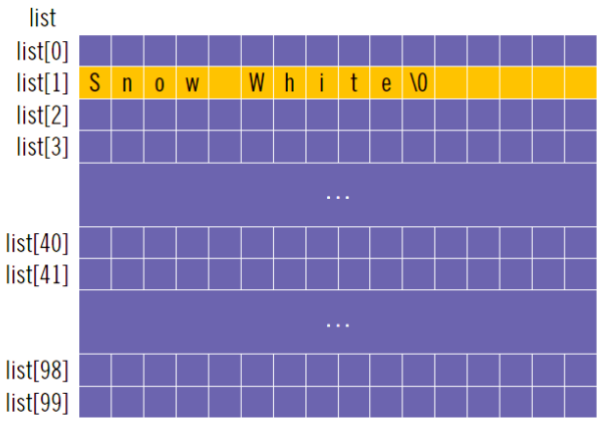


Figure (4-2): Array list, showing list [1]

Suppose that you want to read and store data in list and that there is one entry per line. The following ***for* loop** accomplishes this **inputs** task:

```
for (j = 0; j < 100; j++)
cin.getline(list [j], 16);
```

The following ***for* loop outputs** the string in each row:

```
for (j = 0; j < 100; j++)
cout << list [j] << endl;
```

You can also use other string functions (such as **strcmp** and **strlen**) and ***for*** loops to manipulate list.

L 10 :

Notes:

- ✓ In C++, the **null character** is represented as '**\0**'.
- ✓ From the definition of **C-strings**, it is clear that there is a difference between 'A' and "A". The first one is **character** A; the second is **C-string** A. Because **C-strings** are null-terminated, "A" represents two characters: 'A' and '\0'. Similarly, the **C-string** "Hello" represents **six** characters: 'H', 'e', 'l', 'l', 'o', and '\0'.
- ✓ To store 'A', we need only **one** memory cell of type **char**; to store "A", we need two memory cells of type **char**—one for 'A' and one for '\0'. Similarly, to store the **C-string** "Hello" in computer memory, you need **six** memory cells of type **char**.
- ✓ The length of a **C-string** is the number of actual characters enclosed in double quotation marks; for example, the length of the **C-string** "Hello" is 5.
- ✓ **C-strings** are stored in character arrays.
- ✓ **C-string** index starts with 0. The array's index can be any expression that evaluates to a non-negative integer.(i.e. str[0], str[1], str[i] or str[i+1] , etc.)
- ✓ **C-strings** are null-terminated; that is, the last character in a **C-string** is always the **null character**. Therefore, the **null character** should not appear anywhere in the **C-string** except the **last** position.
- ✓ The only aggregate operations allowed on **C-strings** are input and output. To use other operations, the programmer needs to include the header file cstring (string.h), which contains the specifications of many functions for **C-string** manipulation.
- ✓ C++ provides many (predefined) functions such as **strcpy**, **strlen**, **strcat** and **strcmp** that you can use with **C-strings**. And also provides several (predefined) functions that can help you manipulate individual **characters**. You may have to include the header file cctype (ctype.h).
- ✓ **C-strings** are compared **character-by-character**.

- ✓ Because **C-strings** are stored in arrays, individual characters in the **C-string** can be accessed using the array index (**subscript**) operator [ ].
- ✓ Both the **C-string** library functions and the **String** library functions are available to C++ programs. But, don't forget that these are two different function libraries.
- ✓ In the following examples we attempt to draw the distinction between the two string representations (**C-string** & **String**) and their associated operations.

String (#include <string>)	C-string (#include <cstring>)
Declaring	
`string str;`	`char str[10];`
Initializing	
`string str1 = "Ahmed Ali!";`	`char str1[ ] = "Ahmed Ali!";`
`string str2("Call home!");`	`char str2[11] = "Call home!";`

`string str3("OK");`	`char str3[ ] = {'O', 'K', '\0'};`
	`Last line above has same effect as:`
	`char str3[ ] = "OK";`

Assigning	

`string str;`
`str = "Hello";`
`str = otherString;`
	`Can't do it, i.e., can't do this:`
	`char str[10];`
	`str = "Hello!"; // illegal`

Concatenating	

`str1 += str2; // str1 = str1 + str2;`
`str = str1 + str2;`
	`strcat(str1, str2);`
	`strcpy(str, strcat(str1, str2));`

Copying	

`string str;`
`str = "Hello";`
`str = otherString;`
	`char str[20];`
	`strcpy(str, "Hello");`
	`strcpy(str, otherString);`

Accessing a single character	
`str[index]` `str.at(index)`	`str[index]`

Comparing	

`if (str1 < str2)`
`  cout << "str1 comes 1st.";`
`if (str1 == str2)`
`  cout << "Equal strings.";`
`if (str1 > str2)`
`  cout << "str2 comes 1st.";`
	`if (strcmp(str1, str2) < 0)`
	`  cout << "str1 comes 1st.";`
	`if (strcmp(str1, str2) == 0)`
	`  cout << "Equal strings.";`
	`if (strcmp(str1, str2) > 0)`
	`  cout << "str2 comes 1st.";`

Finding the length	
`str.length();  //  str.size();`	`strlen(str);`
Input	

`cin >> s;`
 	
`getline(cin, s);`
 	
`getline(cin, s, 'x');`
	`cin >> s;`
	
	`cin.getline(s, numCh+1);`
	`cin.getline(s, numCh+1, '\n');`
	
	`cin.getline(s, numCh+1, 'x');`
	In all cases, numCh is the maximum number of characters that will be read.

Output	
`cout << str;`	`cout << str;`