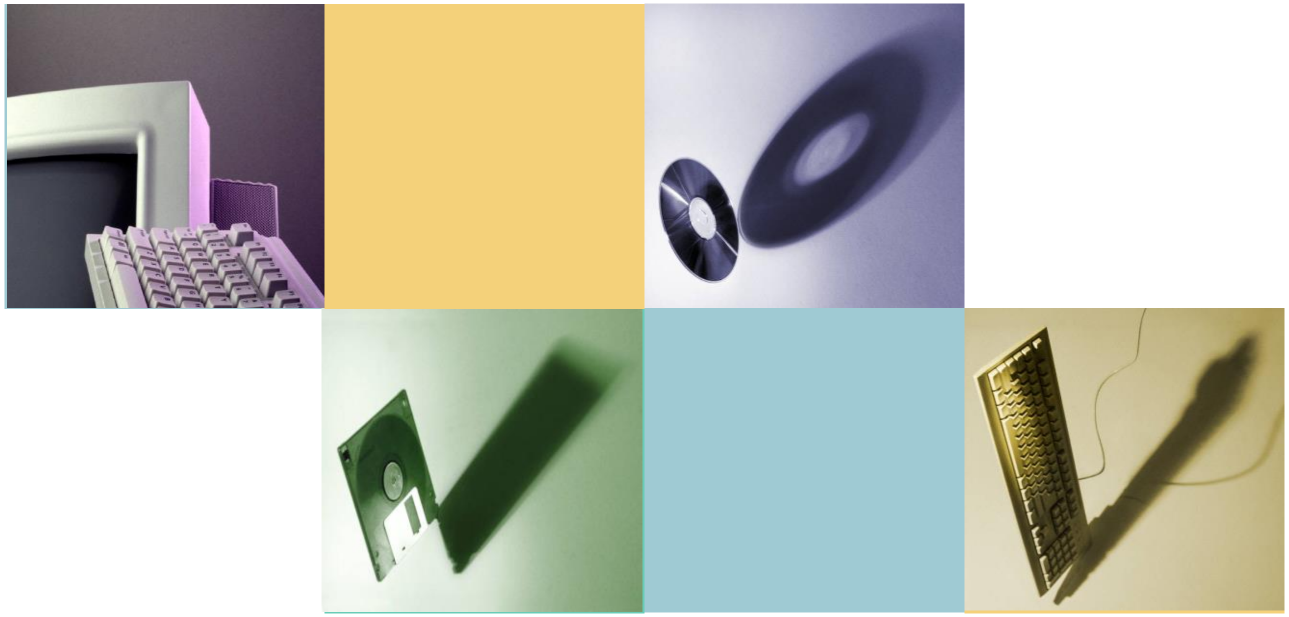
Object-Oriented Programming

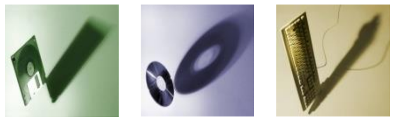


Chuan-Kang Ting

Dept of Computer Science and Information Engineering
National Chung Cheng University

Chapter 9

Strings



Outline

- **C-Strings**

  The way C++ support the old method

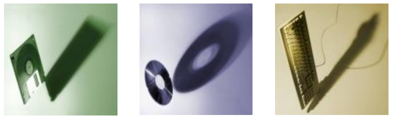
  – An array type for strings

  – Character manipulation tools

- **Strings in C++**

  The C++'s solution to strings

  – The standard class **`string`**



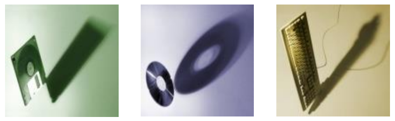
C-Strings?

- **Use**
  - Old fashion from C, but still widely used
  - We've used it
    - e.g. "Hello" → 5 letters + 1 null character
- Pain in using C-strings?

→ **C-string is just an array of characters**

  - Base type **`char`**
  - One character per indexed variable
  - One extra character **'`\0`'**
    - Called null character
    - End marker

Relieve the pain by viewing C-strings as **partially-filled arrays**

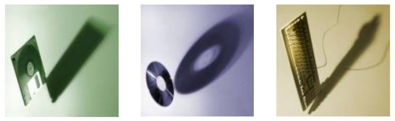


C-String Variables (1)

- **An array of characters**
  - e.g. char s[10];
    - For 9 letters
    - + 1 null character
  - Array → one character per indexed variable
    - If s contains “`Hi mom!`”, the array elements are filled as

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\o	?	?

  - s[0] is ‘H’
  - s[1] is ‘i’
  - …
  - s[7] is ‘\0’
  - s[8] and s[9] are unknown



C-String Variables (2)

- **Syntax**

```
char Array_Name[Max_Size + 1];
```

- **Partially filled array**
  - Partially filled array
    - Uses an int variable, e.g. numberUsed, to keep track how much of the array is used
  - C-string variable
    - Uses the null character ‘\0’ to mark the end of the string

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

C-String Variables (3)

- **Initialization**
  - Need NOT fill the entire array
    - e.g. `char myMessage[20] = "Hi there";`
  - Can omit the size

```
char shortString[4] = "abc";
```

is equivalent to

```
char shortString[] = "abc";
```

  - Places '\0' at end

```
char shortString[] = "abc";
```

is not equivalent to

```
char shortString[] = {'a', 'b', 'c'};
```

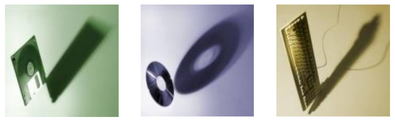
Use '**\0**' for **partially filled array**

C-String Variables (4)

- **C-string index manipulation**
  - As if manipulating indexed variables of an array
  - Be careful with ‘\0’
  - If the array loses ‘\0’, it no longer behaves like a C-string variable
    - Unpredictable results
  - e.g.

```
char happyString[7] = “DoBeDo”;
happyString[6] = ‘Z’;
```

‘\0’ was overwritten!



C-String Variables (5)

- **Using = and ==**
  - A C-string is an **array** of characters, NOT data type
    → Many of usual operations do not work
  - **Assignment** statement, =
    - The assignment does NOT work (why?)

```
char aString[10];
aString = "Hello";   //illegal!
```

For an array, only assignment to *individual* elements is allowed

Instead, use the function **`strcpy`**

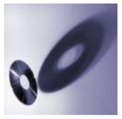
```
strcpy(aString, "Hello");
```

Do an array-like copy

- But this works! (why?)

```
char happyString[7] = "DoBeDo";
```

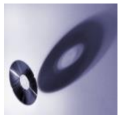
  - (Recall: The use of equal sign in a declaration is an **initialization**, not an assignment)



C-String Variables (6)

- **Using = and ==** (cont'd)
  - **Comparison**, ==
    - Cannot use in C-strings (why?)
      - Incorrect results with no error message!
    - Instead, use the function **`strcmp`**
      - e.g. `if (strcmp(cString1, cString2))`…
      - Returns the so-called lexicographic order
        - » If the C-strings are the same, returns *0* (false)
        - » If cString1 < cString2 in lexicographic order, return a *negative* number (true)
        - » If cString1 > cString2 in lexicographic order, return a *positive* number (true)

Compare elements of two arrays

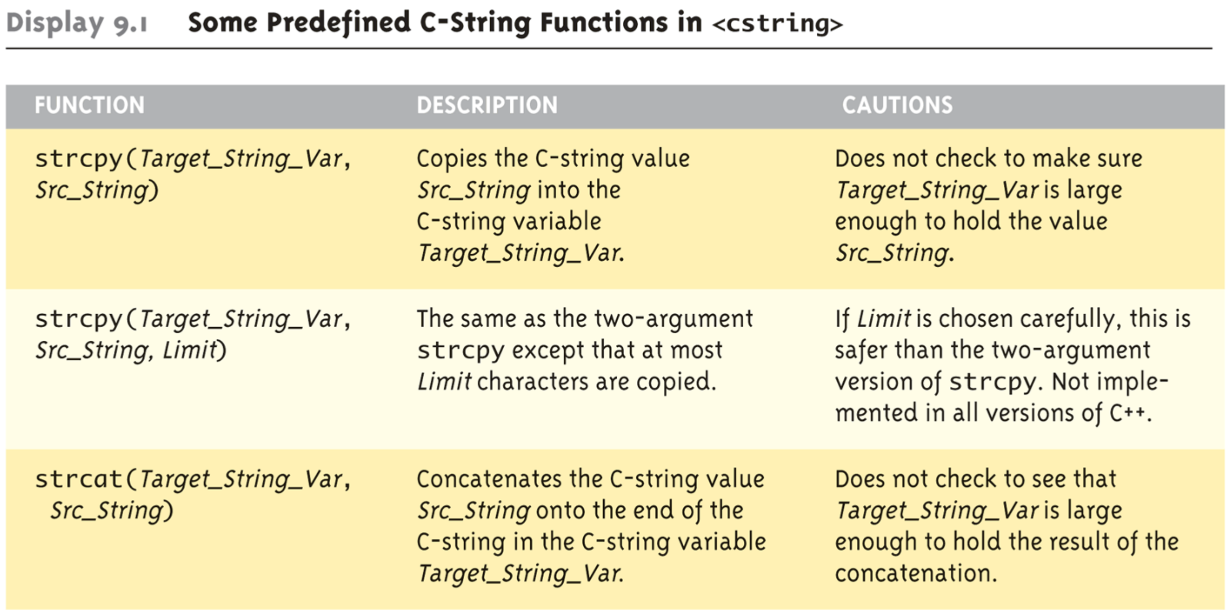


The <cstring> Library (1)

- **Use**

```
#include <cstring>
```

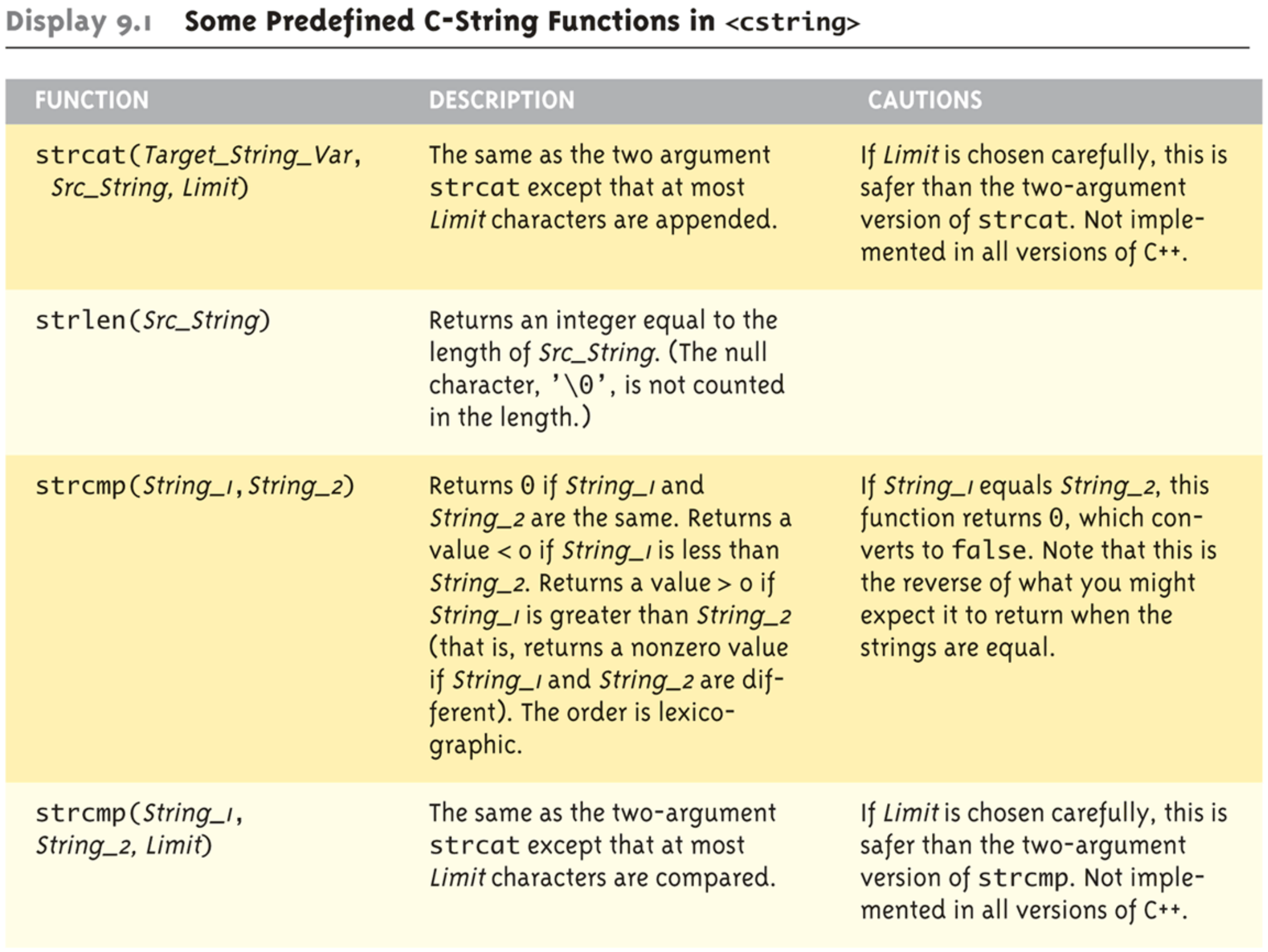
  - In the *global* namespace, not in the `std` namespace → need NOT `using` statement



Display 9.1 **Some Predefined C-String Functions in `<cstring>`**

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy(`*Target_String_Var*, *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy(`*Target_String_Var*, *Src_String*, *Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*, *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

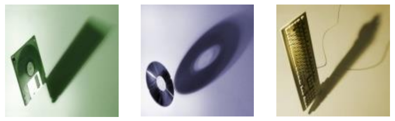
The <cstring> Library (2)


Display 9.1 **Some Predefined C-String Functions in** `<cstring>`

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*`,` *Src_String, Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*`,`*String_2*`)`	Returns `0` if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns `0`, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*`,` *String_2, Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

C-String Arguments and Parameters

- **C-string is an array → C-string parameter is array parameter**
  - C-strings passed to functions can be changed by receiving function
  - Send the size of C-string
    - by explicit indication, as used in arrays
    - by detecting the null character ‘\0’
  - Use “const" modifier to protect c-string arguments

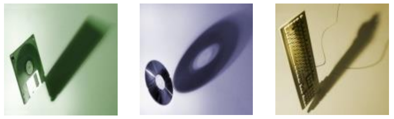


C-string Input & Output (1)

- **Output operator <<**
  - Works well
    - Because << is overloaded for C-string
- **Input operator >>**
  - Works, but with some problems
    - Whitespace (blanks, tabs, and line breaks) are delimiter
    - Delimiter are **skipped**
    - Reading of input **stops** at delimiter

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << b << "END\n";
```

```
Enter some input:
Do be do to you!
DobeEND
```



C-string Input & Output (2)

- **Function `getline`**
  - A member function of every input stream
    - e.g. cin or a file input stream
  - Receives entire line into c-string
  - Can explicit tell the length to receive
  - e.g.

```
char shortString[5];
cout << "Enter some input:\n";
cin.getline(shortString, 5);
cout << shortString << "END\n";
```

```
Enter some input:
Do bedowap
Do bEND
```

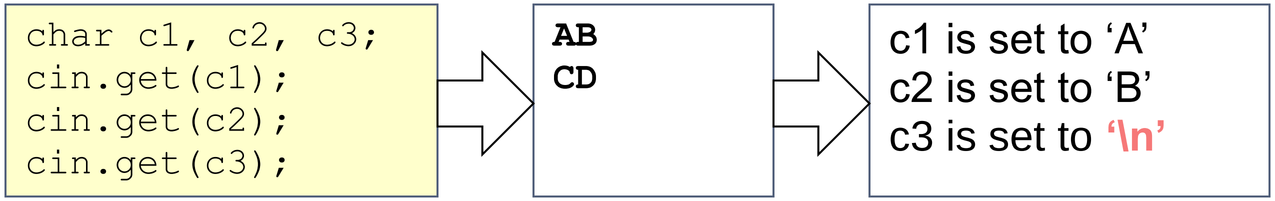
4 (not 5) characters are read (Why?)

Character I/O

- **Input and output data**
  - All treated as character data
  - e.g. *number* 10 is outputted as two *characters* ‘1’ and ‘0’
  - Conversion done automatically
- **But…**
  - Sometimes the conversion gets in the way
  - C++ provides some low-level facilities for character I/O
  - Converse data yourselves

get and put (1)

- **`get`**
  - Reads one char at a time
  - Every input stream has **`get`** as a member function
  - Can read *any* character, including whitespace
  - Useful to detect the end of a line



```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

```
AB
CD
```

c1 is set to ‘A’
c2 is set to ‘B’
c3 is set to **‘\n’**

get and put (2)

- **`put`**
  - Outputs one char at a time
  - Every output stream has **`put`** as a member function
  - Can output *any* character

(→ Do nothing more than cout, but can be useful in file I/O)

```
cout.put('a');
cout.put("a");
```

'a': one char
"a": one string (plus '\0')

```
Error E2034 test4.cpp 7: Cannot convert 'char *' to 'char' in
function main()
Error E2342 test4.cpp 7: Type mismatch in parameter '__c'
(wanted 'char', got 'char *') in function main()
```

Unexpected ‘\n’ in Input

- **Leftover ‘\n’**
  - A common problem of forgetting to remove the ‘\n’ that ends every input line

```
cout << “Enter a number:\n”;
int number;
cin >> number;
cout << “Now enter a letter:\n”
char symbol;
cin.get(symbol);
```

```
Enter a number:
21
Now enter a letter:
A
```

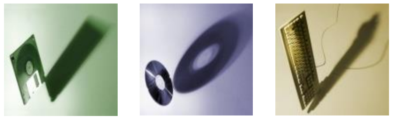
number will be **21**
symbol will be **‘\n’**

→ **`cin` lefts ‘\n’,** while `get` does NOT skip over whitespace

```
char c;
do {
  cin.get(c);
} while(c != ‘\n’);
```

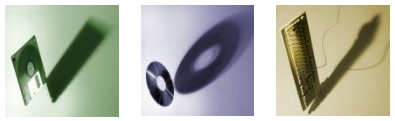
OR

```
cin >> symbol;
```



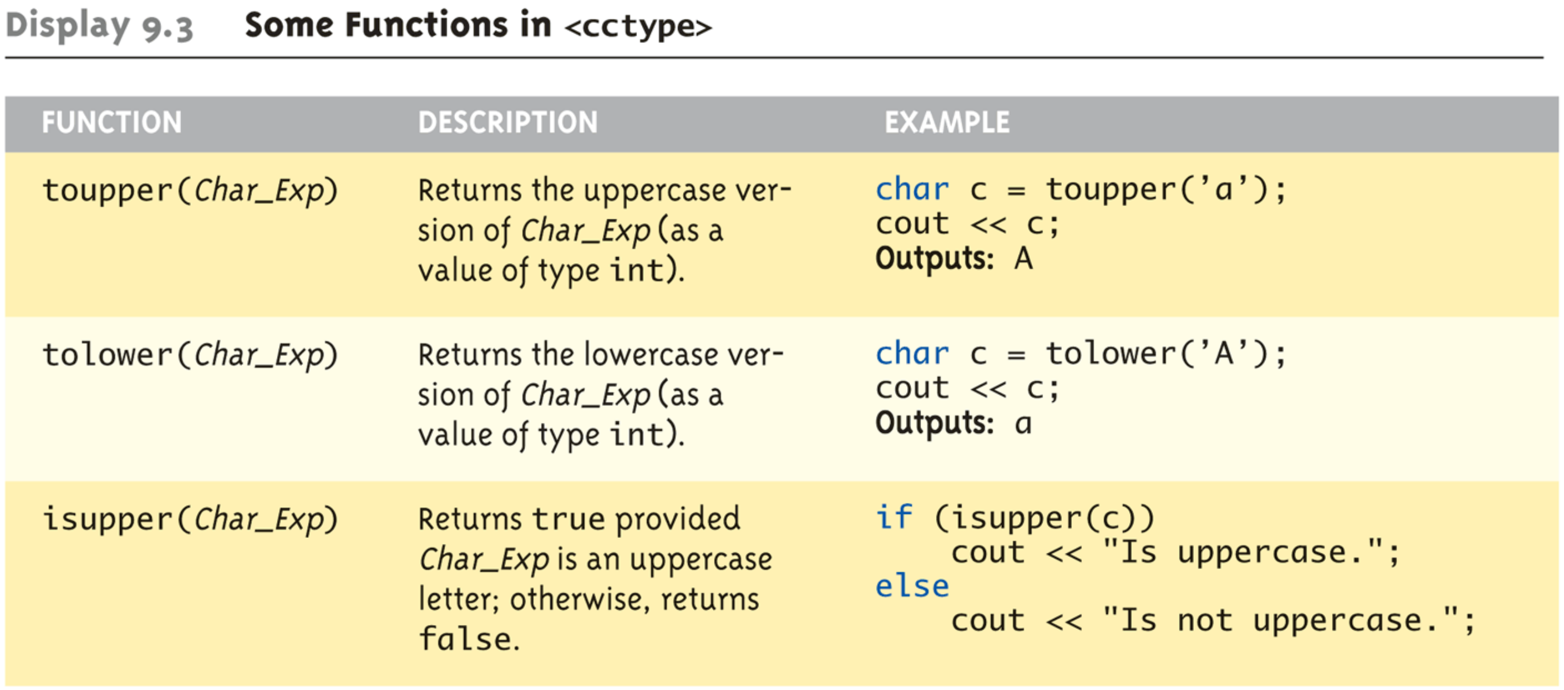
More Member Functions

- **`putback()`**
  - Places one char back in the input stream
  - `cin.putback(nextCharToReadIn);`
- **`peek()`**
  - Returns next char, but leaves it there
  - `peekChar = cin.peek();`
- **`ignore()`**
  - Skip input, up to designated character
  - `cin.ignore(1000, '\n');`
    - Skips at most 1000 characters until '\n'



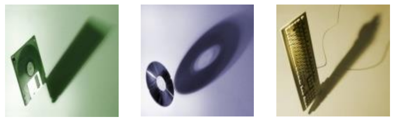
Character-Manipulating Functions (1)

- **Regular functions**
  - instead of member functions (of cin)

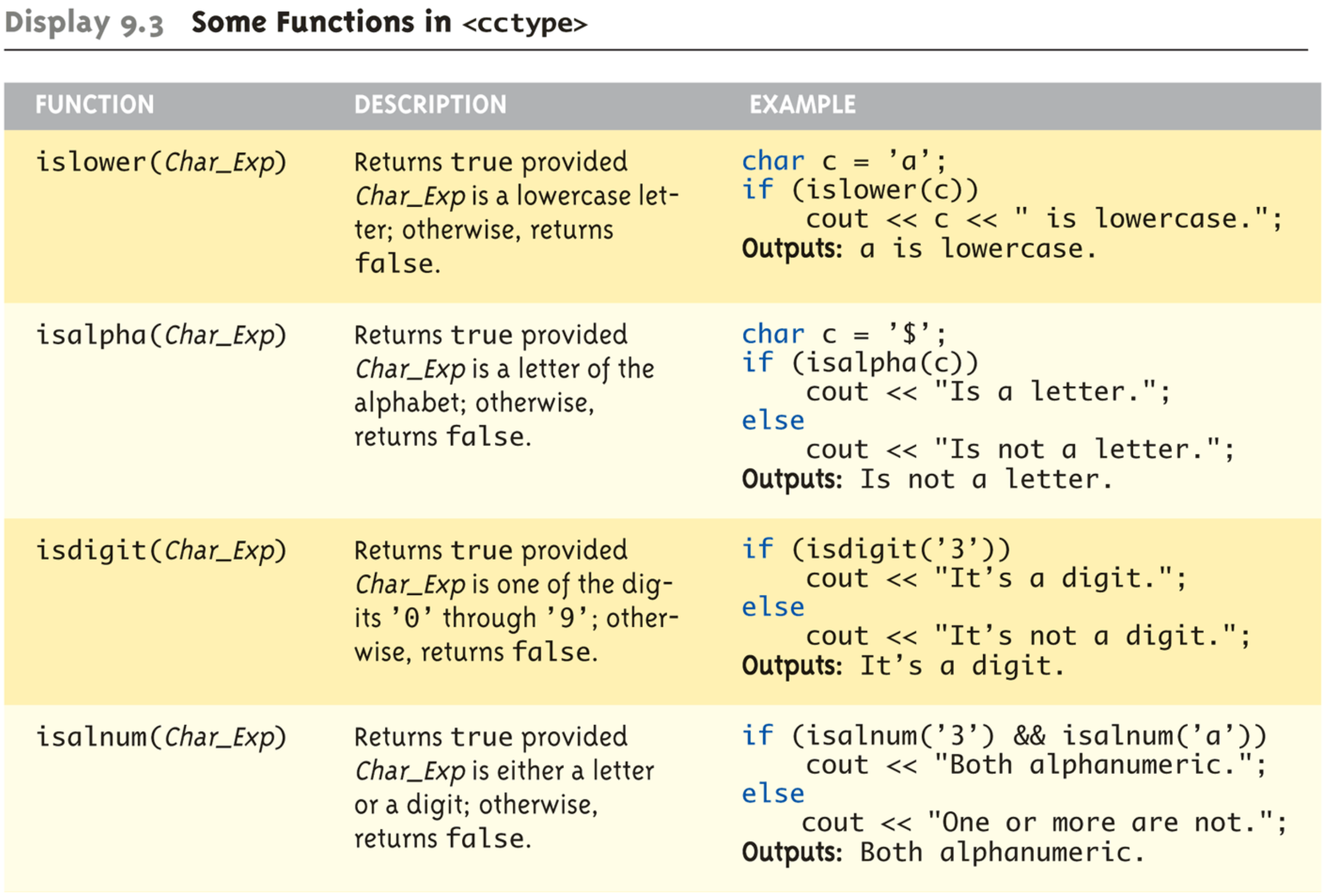


Display 9.3 **Some Functions in <cctype>**

FUNCTION	DESCRIPTION	EXAMPLE
`toupper(`*Char_Exp*`)`	Returns the uppercase version of *Char_Exp* (as a value of type `int`).	`char c = toupper('a');` `cout << c;` **Outputs:** A
`tolower(`*Char_Exp*`)`	Returns the lowercase version of *Char_Exp* (as a value of type `int`).	`char c = tolower('A');` `cout << c;` **Outputs:** a
`isupper(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is an uppercase letter; otherwise, returns `false`.	`if (isupper(c))` `cout << "Is uppercase.";` `else` `cout << "Is not uppercase.";`



Character-Manipulating Functions (2)



Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE

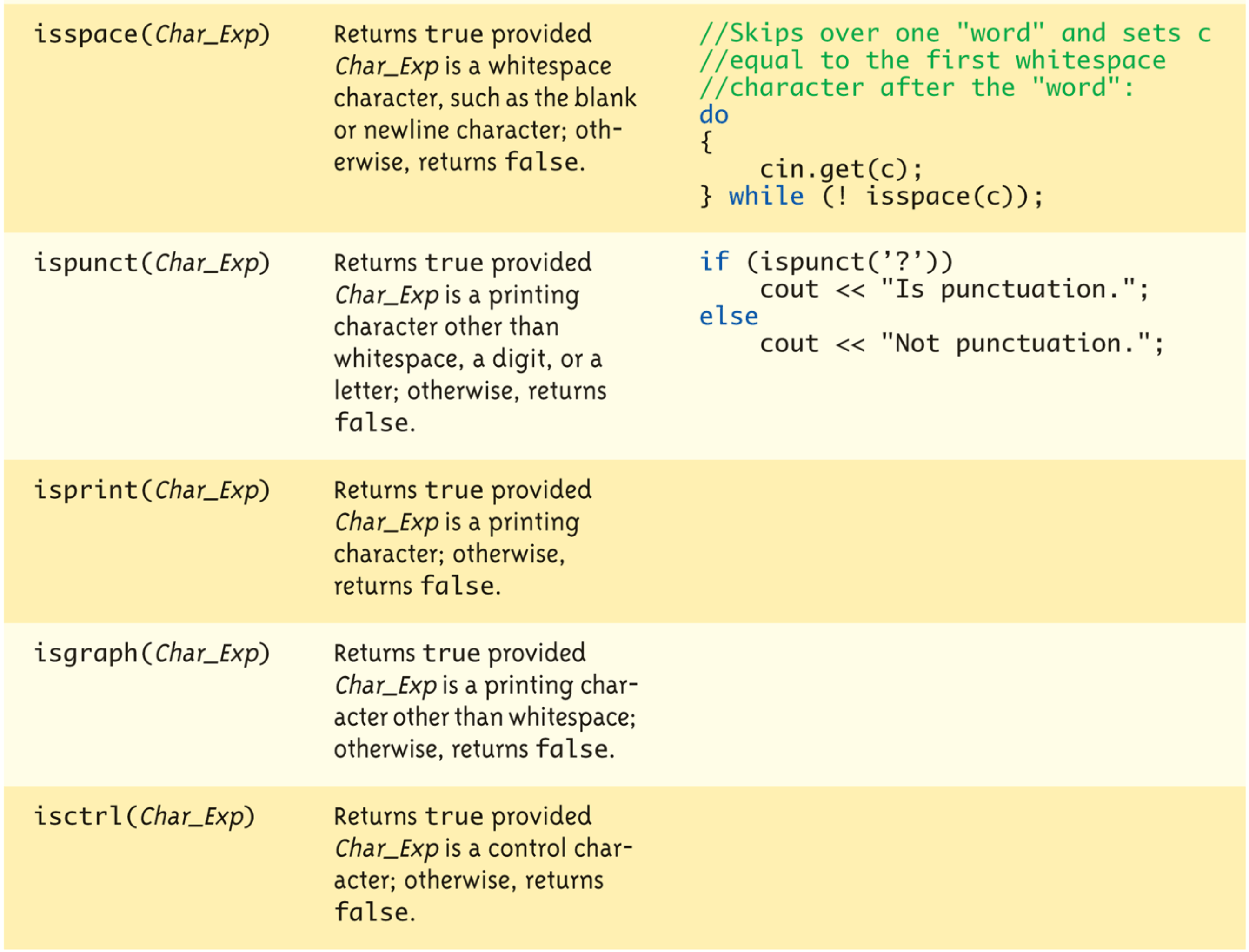
`islower(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a lowercase letter; otherwise, returns `false`.	`char c = 'a';`
		`if (islower(c))`
		`    cout << c << " is lowercase.";`
		Outputs: `a is lowercase.`

`isalpha(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a letter of the alphabet; otherwise, returns `false`.	`char c = '$';`
		`if (isalpha(c))`
		`    cout << "Is a letter.";`
		`else`
		`    cout << "Is not a letter.";`
		Outputs: `Is not a letter.`

`isdigit(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is one of the digits `'0'` through `'9'`; otherwise, returns `false`.	`if (isdigit('3'))`
		`    cout << "It's a digit.";`
		`else`
		`    cout << "It's not a digit.";`
		Outputs: `It's a digit.`

`isalnum(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is either a letter or a digit; otherwise, returns `false`.	`if (isalnum('3') && isalnum('a'))`
		`    cout << "Both alphanumeric.";`
		`else`
		`    cout << "One or more are not.";`
		Outputs: `Both alphanumeric.`

Character-Manipulating Functions (3)



`isspace(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a whitespace character, such as the blank or newline character; otherwise, returns `false`.	`//Skips over one "word" and sets c`
		`//equal to the first whitespace`
		`//character after the "word":`
		`do`
		`{`
		`    cin.get(c);`
		`} while (! isspace(c));`

`ispunct(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace, a digit, or a letter; otherwise, returns `false`.	`if (ispunct('?'))`
		`    cout << "Is punctuation.";`
		`else`
		`    cout << "Not punctuation.";`

`isprint(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character; otherwise, returns `false`.	
`isgraph(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace; otherwise, returns `false`.	
`isctrl(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a control character; otherwise, returns `false`.	

Standard Class `string`

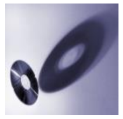
- **Class string**
  - Treats string as a basic data type

    (Recall: C-strings are *arrays* of char with '\0')
  - To use

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

✓ – Supports =, ==, +

```
string s3;               //default constructor: initializes a empty string
string s1("Mid");        //constructor: convert C-string to string (no '\0')
string s2 = "term";      //equivalent to string s2("term");
s3 = s1 + s2;            //assignment(=), add(+)
```



An Example

Display 9.4 Program Using the Class `string`

```cpp
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

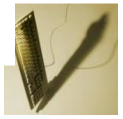
Initialized to the empty string.

Two equivalent ways of initializing a string variable

Overloading +

Sample Dialogue

```
I love fried ants!
Bon appetite!
```



I/O with `string` (1)

- **Just like other types!**
  - `cin` and `cout`

```
string s1, s2;
cin >> s1;
cin >> s2;
```

```
May the force be with you!
```

s1 is "**May**";
s2 is "**the**"

→ The extraction operator `cin` reads in words

- Reading of input stops at delimiter (whitespace)

I/O with `string` (1)

- **Function `getline`**
  - Reads an entire line of input into string
  - Not a member function
  - Syntax: getline(*io_stream*, *string*) ← Default: '\n'

    getline(*io_stream*, *string, stopping_delimiter*)
  - e.g.

```
string line1, line2;
cout << Enter two lines of input:\n";
getline(cin, line1);
getline(cin, line2, '!');
cout << line1 << "--Joda\n";
cout << line2 << "--Luke Skywalker\n";
```

```
Enter two lines of input:
May the force be with you!
Thanks! Jedi Master
May the force be with you!--Joda
Thanks--Luke Skywalker
```

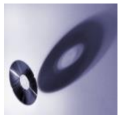
without '!'

Processing with `string`

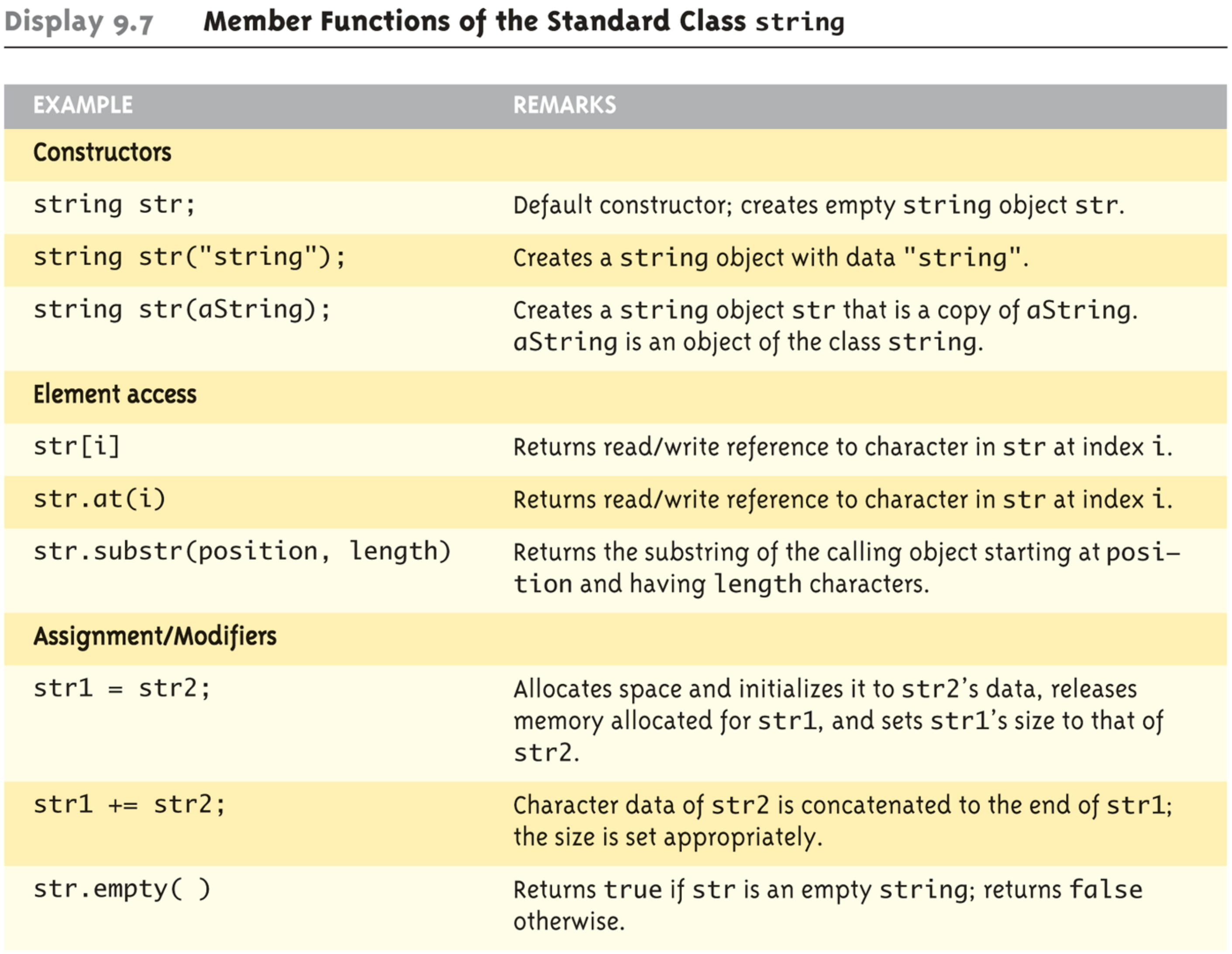
- **Same operations available as C-strings**
  - In the same way of accessing array element
  - e.g. `lastName[i]` for a string object lastName
- **And more**
  - Over 100 members of standard string class
- **Some member functions**

  `.length()`
  - Returns the length of string variable

  `.at(i)`
  - Similar to lastName[i] but, moreover, it checks if the index i is legal



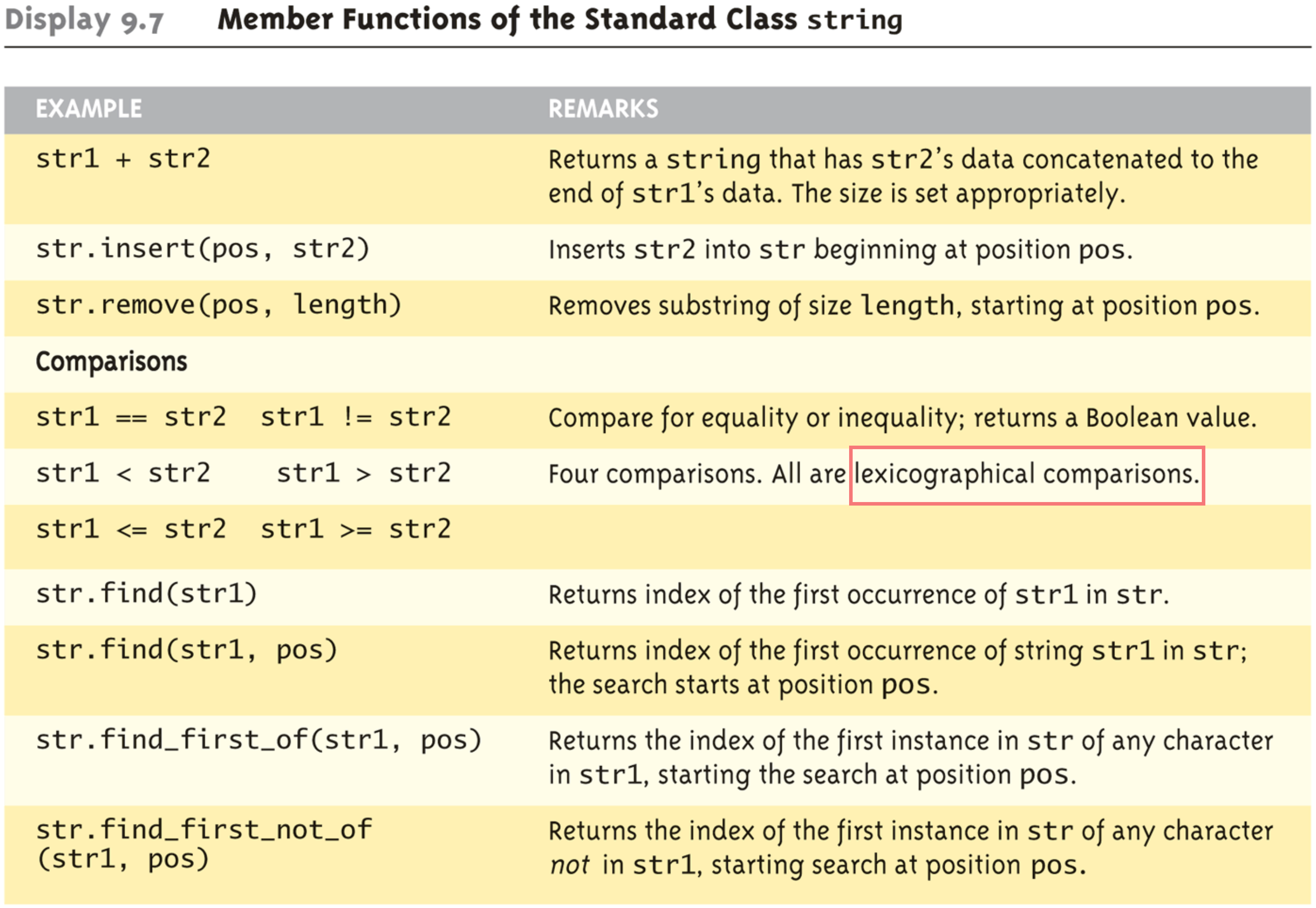
Some Member Functions of `string` (1)



Display 9.7 **Member Functions of the Standard Class `string`**

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.

Some Member Functions of `string` (2)



Display 9.7 **Member Functions of the Standard Class `string`**

EXAMPLE	REMARKS
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position `pos`.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position `pos`.
Comparisons	
`str1 == str2` `str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2` `str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2` `str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position `pos`.

C-String ←→ `string` Object

- **Automatic type conversion**
  - C-string → string object
    - Perfectly legal and appropriate!

```
char aCString[] = "My C-string";
string stringVar;
stringVar = aCstring;
```

How to make it?
→ overloading assignment

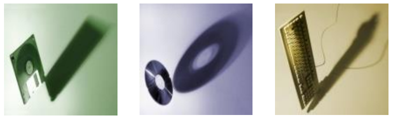
  - C-string ← string object
    - No auto-conversion of string object to C-string

```
aCString = stringVar;     //Illegal
```

    - Must use explicit conversion

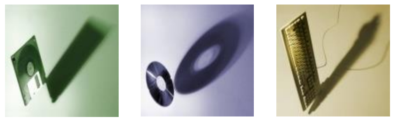
```
strcpy(aCString, stringVar.c_str());
```

returns the corresponding C-string



Summary (1)

- **C-String**
  - Array of characters plus ‘\0’
  - Libraries <cstring> and <cctype> have useful manipulating functions
  - cin and cout
    - The extraction operator >> ignores whitespace
    - cin.getline
    - cin.get(c)
    - cout.put(c)



Summary (2)

- **Standard Class `string`**
  - Treated as a basic data type
  - Better behaved than C-strings
    - Supports =, ==, +
    - Lots of useful member functions
  - Conversion between C-strings and string objects
    - C-string → string object: Automatic
    - String object → C-string: Manual