

# **Tools for I/O**

## **String & Character Manipulators**

**CS 16: Solving Problems with Computers I**  
**Lecture #11**

Ziad Matni  
Dept. of Computer Science, UCSB

# Announcements

---

- **Homework #10 due today**
- **Lab #6 is due on Friday at Noon**

# Lecture Outline

---

- Formatting output
- Character manipulators
- Character I/O
- String manipulators

# Tools for Stream I/O

---

- Formatting a program's output:
  - The spaces between items
  - The number of digits after a decimal point
  - The numeric style: scientific notation for fixed point
  - Showing digits after a decimal point even if they are zeroes
  - Showing plus signs in front of positive numbers
  - Left or right justifying numbers in a given space

# Formatting Output to Files

---

- Format output to the screen with:

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

- Format output to a file using `out_stream` with:

```
out_stream.setf(ios::fixed);  
out_stream.setf(ios::showpoint);  
out_stream.precision(2);
```



# precision(*n*);

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

- **precision** is a member function of output streams

- After `out_stream.precision(2);`

the output of numbers with decimal points will show:

- a total of 2 *significant digits*

23. 2.2e7 2.2 6.9e-10.00069

OR

- 2 *digits after the decimal point*

23.56 2.26e7 2.21 0.69 0.69e-4

*n significant digits*  
vs  
*n digits after the decimal pt*

- Calls to **precision** apply  
only to the stream named in the call

# setf(ios::fixed);

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

- **setf** is a member function of output streams
  - setf is an abbreviation for *set flags*
    - **ios::fixed** is a formatting flag
  - `out_stream.setf(ios::fixed);`  
All further output of floating point numbers are written in *fixed-point notation*
  - There are other formatting flags for setf
- Calls to **setf** apply  
only to the stream named in the call

# setf(ios::showpoint);

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

After `out_stream.setf(ios::showpoint);`

output of floating point numbers  
shows the decimal point  
even if all digits after the decimal point are  
zeroes



### Formatting Flags for setf

| Flag                         | Meaning   | Default |
|------------------------------|---|---------|
| <code>ios::fixed</code>      | If this flag is set, floating-point numbers are not written in e-notation. (Setting this flag automatically unsets the flag <code>ios::scientific</code> .)   | Not set |
| <code>ios::scientific</code> | If this flag is set, floating-point numbers are written in e-notation. (Setting this flag automatically unsets the flag <code>ios::fixed</code> .)<br>If neither <code>ios::fixed</code> nor <code>ios::scientific</code> is set, then the system decides how to output each number.  | Not set |
| <code>ios::showpoint</code>  | If this flag is set, a decimal point and trailing zeros are always shown for floating-point numbers. If it is not set, a number with all zeros after the decimal point might be output without the decimal point and following zeros.   | Not set |
| <code>ios::showpos</code>    | If this flag is set, a plus sign is output before positive integer values.  | Not set |
| <code>ios::right</code>      | If this flag is set and some field-width value is given with a call to the member function <code>width</code> , then the next item output will be at the right end of the space specified by <code>width</code> . In other words, any extra blanks are placed <i>before</i> the item output. (Setting this flag automatically unsets the flag <code>ios::left</code> .) | Set     |
| <code>ios::left</code>       | If this flag is set and some field-width value is given with a call to the member function <code>width</code> , then the next item output will be at the left end of the space specified by <code>width</code> . In other words, any extra blanks are placed <i>after</i> the item output. (Setting this flag automatically unsets the flag <code>ios::right</code> .)  | Not set |

# Creating Space in Output

- The **width member** function specifies the number of spaces for the next item
  - Applies *only to the next item of output*

## **Example:**

- To print the digit 7 in four spaces and use

```
out_stream.width(4);  
out_stream << 7 << endl;
```

Three of the spaces will be blank:



`.setf(ios::right)`  
*default*



`.setf(ios::left)`

# Not Enough Width?

---

- What if the argument for width is too small?
  - Such as specifying `cout.width(3);` when the value to print is **3456.45**
- The entire item is always put in output
  - If too few spaces are specified, as many more spaces as needed are used
  - In the example above, the value is still printed as if the `cout.width(3);` was not there.

# Unsetting Flags

---

- Any flag that is set, may be unset
- Use the **unsetf** function

– Example:

```
cout.unsetf(ios::showpos);
```

causes the program to stop printing  
plus signs on positive numbers



# Manipulators

---

- A function called in a nontraditional way
  - Manipulators, in turn, call member functions
  - Manipulators may or may not have arguments
  - Used after the insertion operator (<<) as if the manipulator function call is an output item

# The **setw** Manipulator

- **setw** does the same task as member function **width**
  - **setw** calls the width function to set spaces for output
  - Found in the library **<iomanip>**
- Example: 

```
cout << "Start" << setw(4) << 10  
      << setw(4) << 20 << setw(6) << 30;
```

produces:    Start    10    20    30

4 Spaces                  6 Spaces

# The `setprecision` Manipulator

- `setprecision` does the same task as member function `precision`
  - Found in the library `<iomanip>`
- Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2)
    << 10.3 << endl
    << "$" << 20.5 << endl;
```

produces:   \$10.30  
              \$20.50
- `setprecision` setting stays in effect until changed

# More on File I/O

---



# Stream Names as Arguments

---

- Streams can be arguments to a function
  - The function's formal parameter for the stream must be call-by-reference

- Example:

```
void make_neat(ifstream& messy_file,  
               ofstream& neat_file);
```

# Detecting the End of a File

- Input files used by a program may vary in length
  - Programs may not be able to assume the number of items in the file
- 2 ways to know if the end of the file is reached:
  - The Boolean expression `(in_stream.eof())`
    - Utilizes the member function `eof()` ... or end-of-file
    - **True** if you have reached the end of file
    - **False** if you have not reached the end of file
  - The Boolean expression `(in_stream >> next)`
    - Reads a value from `in_stream` and stores it in `next`
    - **True** if a value *can* be read and stored in `next`
    - **False** if *there is not a value to be read* (i.e. the end of the file)

# End of File Example

*using while (ifstream >> next) method*

- To calculate the average of the numbers in a file that contains numbers of type double:

```
double next, sum = 0;
int count = 0;
while(in_stream >> next) {
    sum = sum + next;
    count++;
}
double average = sum / count;
```

# End of File Example

*using `while ( !ifstream.eof() )` method*

---

- To read each character in a file,  
and then write it to the screen:

```
in_stream.get(next);  
while ( ! in_stream.eof( ) ) {  
    cout << next;  
    in_stream.get(next);  
}
```



# Which of the 2 Should I Use?!

---

In general:

- Use **eof** when input is treated as text and using a member function `get` to read input
- Use the **extraction operator** method when processing numerical data

# Stream Arguments and Namespaces

---

- Using directives have been local to function definitions in the examples so far
- When parameter type names are in a namespace, a **using directive** must be *outside the function* so that C++ will understand the parameter type names such as *ifstream*
  - Using directive example: `using namespace std;`
- Easy solution: place the using directive at the start of the file
  - Many experts do not approve of this,  
because it does not allow for using  
multiple namespaces with names in common



# Character I/O

---

## All data is input and output as characters

- Output of the number 10 is two characters '1' and '0'
- Input of the number 10 is also done as '1' and '0'
- Interpretation of 10 as the number 10  
or as 2 characters depends on the program
- Conversion between characters and numbers  
is *usually* automatic, but *not always*



# Low Level Character I/O

---

Low level C++ functions for character I/O:

- Perform character input and output
- Do **not** perform automatic conversions
- Allow you to do I/O in anyway you can devise

# Member Function `get(char)`

---

- Member function of every input stream
  - i.e. works for `cin` and for `ifstream`
- Reads ***one character*** from an input stream
- Stores the character read in a variable of **type `char`**, which is the single argument the function takes
- Does not use the extraction operator (`>>`)
  - `>>` actually performs some other automatic work
- Does not skip blanks, tabs, new lines
  - These are characters too!

# Using **get**

---

- These lines use **get** to read a character and store it in the variable *next\_symbol*

```
char next_symbol;  
cin.get(next_symbol);
```

- Any character will be read with these statements
  - Blank spaces too!
  - '\n' too! (The newline character)

# get Syntax

---

- `input_stream.get(char_variable);`

- Examples:

```
char next_symbol;  
cin.get(next_symbol);
```

```
ifstream in_stream;  
in_stream.open("infile.dat");  
in_stream.get(next_symbol);
```



# More About get

- Given this code:

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

and this input: **AB**  
**CD**

- `c1 = 'A'`   `c2 = 'B'`   `c3 = '\n'`
- On the other hand: `cin >> c1 >> c2 >> c3;`  
would place '**C**' in c3 because ">>" operator  
skips newline characters

# The End of The Line using `get`

- To read and echo an entire line of input by collecting all characters before the newline character

- Look for '`\n`' at the end of the input line:

```
cout << "Enter a line of input and I will "
      << "echo it.\n";
char symbol;
do {
    cin.get(symbol);
    cout << symbol;
} while (symbol != '\n');
```

- All characters, including '`\n`' will be output

## NOTE: '\n ' vs "\n "

---

- '\n'
  - A value of type char
  - **Can** be stored in a variable of type char
- "\n"
  - A string containing only one character
  - **Cannot** be stored in a variable of type char
- In a **cout** statement they produce the same result

# Member Function **put**

---

- Member function of every output stream
  - i.e. works for cout and for ofstream
- Requires **one argument of type char**
- Places its argument of type char in the output stream
- Does not do allow you to do more than previous output with the insertion operator and cout



# put Syntax

---

- **output\_stream.put(char\_variable);**
- Examples:

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

```
ofstream out_stream;  
out_stream.open("outfile.dat");  
out_stream.put('Z');
```

# Member Function **putback**

---

- The **putback** member function puts a char in the input stream
- putback is a member function of every input stream
  - cin, ifstream
- **Useful when input continues until a specific character is read, but you do not want to process that character**
- Character placed in the stream does not have to be a character read from the stream

# putback Example

- The following code reads up to the first blank in the input stream *fin*, and writes the characters to the file connected to the output stream *fout*

```
fin.get(next);  
while (next != ' ' )  
{  
    fout.put(next);  
    fin.get(next);  
}  
fin.putback(next);
```

- The blank space read to end the loop  
is put back into the input stream

# Program Example: Editing a Text File

---

**DEMO!**



# Character Functions

---

- Several predefined functions exist to facilitate working with characters
- The **cctype** library is required for most of them

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

# The **toupper** Function

---

- **toupper** returns the argument's upper case character
  - `toupper('a')` returns 'A'
  - `toupper('A')` return 'A'

# The **tolower** Function

---

- Similar to **toupper** function...
- **tolower** returns the argument's lower case character
  - `tolower('a')` returns 'a'
  - `tolower('A')` return 'a'

# toupper & tolower return int

---

- Characters are actually stored as an integer assigned to the character
- **toupper** and **tolower** actually return the integer representing the character

```
cout << toupper('a');    // prints the integer for 'A' (65)
char c = toupper('a');    // places the integer for 'A' in c
cout << c;                // prints 'A'
cout << static_cast<char>(toupper('a')); // works too
```



# The **isspace** Function

- **isspace** returns *true* if the argument is whitespace
  - Whitespace is: spaces, tabs, and newlines
    - So, **isspace(' ')** returns true, so does **isspace('\n')**
  - Example:

```
if (isspace(next) )  
    cout << '-';  
else  
    cout << next;
```

Prints a '-' if next contains a space, tab, or newline character

### Some Predefined Character Functions in ctype (part 2 of 2)

| Function                       | Description   | Example  |
|--------------------------------|---|--|
| <code>isupper(Char_Exp)</code> | Returns <i>true</i> provided <i>Char_Exp</i> is an uppercase letter; otherwise, returns <i>false</i> .  | <pre>if (isupper(c))     cout &lt;&lt; c &lt;&lt; " is uppercase."; else     cout &lt;&lt; c         &lt;&lt; " is not uppercase.";</pre>  |
| <code>islower(Char_Exp)</code> | Returns <i>true</i> provided <i>Char_Exp</i> is a lowercase letter; otherwise, returns <i>false</i> .   | <pre>char c = 'a'; if (islower(c))     cout &lt;&lt; c &lt;&lt; " is lowercase.";</pre> <b>Outputs:</b> a is lowercase.  |
| <code>isalpha(Char_Exp)</code> | Returns <i>true</i> provided <i>Char_Exp</i> is a letter of the alphabet; otherwise, returns <i>false</i> .                                     | <pre>char c = '\$'; if (isalpha(c))     cout &lt;&lt; c &lt;&lt; " is a letter."; else     cout &lt;&lt; c         &lt;&lt; " is not a letter.";</pre> <b>Outputs:</b> \$ is not a letter. |
| <code>isdigit(Char_Exp)</code> | Returns <i>true</i> provided <i>Char_Exp</i> is one of the digits '0' through '9'; otherwise, returns <i>false</i> .                            | <pre>if (isdigit('3'))     cout &lt;&lt; "It's a digit."; else     cout &lt;&lt; "It's not a digit.";</pre> <b>Outputs:</b> It's a digit.  |
| <code>isspace(Char_Exp)</code> | Returns <i>true</i> provided <i>Char_Exp</i> is a whitespace character, such as the blank or new-line symbol; otherwise, returns <i>false</i> . | <pre>//Skips over one "word" and //sets c equal to the first //whitespace character after //the "word": do {     cin.get(c); } while (! isspace(c));</pre>                                 |

# Program Example:

## Looking for numbers

---

**DEMO!**

# Strings in C++

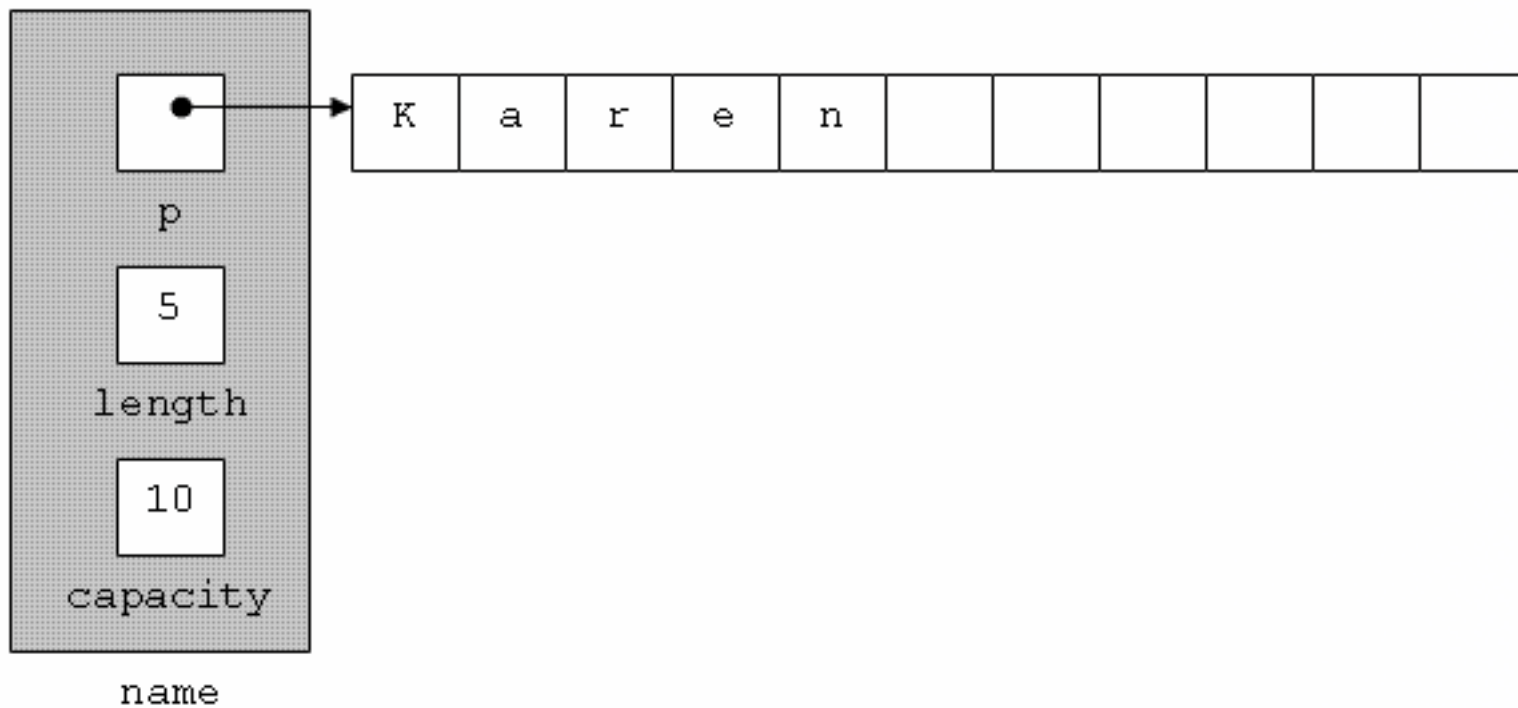
## *A high-level view*

---

- Strings, as used with the `<string>` library, allows the programmer to use strings as a basic data type
- The class of strings are defined as arrays of characters



# The Standard string Class



# String Basics

- Include the `<string>` library
  - There are variable types called **C-strings** as well... more on those later
- Use the `+` operator to concatenate 2 strings

```
string str1 = "Hello ", str2 = "world!", str3;  
str3 = str1 + str2;    // str3 will be "Hello world!"
```
- Use the `+=` operator to append to a string

```
str1 += "Z";    // str1 will be "Hello Z"
```
- Call out a character in the string based on **position**
  - Recall array indices in C++ start at zero (0)

```
cout << str1[0];    // prints out 'H'  
cout << str2[3];    // prints out 'l'
```

# Character Manipulators Work Too!

- Include **<cctype>** to use with, for example, **toupper()**

```
string str1 = "hello";  
str1[0] = toupper(str1[0]);  
cout << str1;    // Will display "Hello"
```

- ...or to use with **tolower()**

```
string str1 = "HeLLo";  
for (int i=0; i < 5; i++)  
    str1[i] = tolower(str1[i]);  
cout << str1;    // Will display "hello"
```

# Built-In String Manipulators

---


- Search functions
  - find, rfind, find\_first\_of, find\_first\_not\_of
- Descriptor functions
  - length, size
- Content changers
  - substr, replace, append, insert, erase



# Search Functions 1

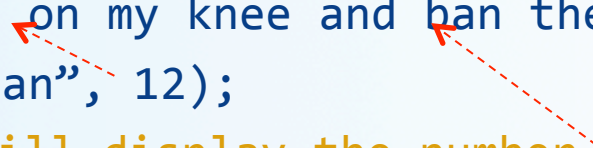
- You can search for the **first occurrence** of a string in a string with the **.find** function

```
string str = "With a banjo on my knee and ban the bomb!";  
int position = str.find("ban");  
cout << position;    // Will display the number 7
```



- You can also search for the **first occurrence** of a string in a string, starting at position **n**

```
string str = "With a banjo on my knee and ban the bomb!";  
int position = str.find("ban", 12);  
cout << position;    // Will display the number 24
```



# Search Functions 2

- You can use the **find** function to make sure a substring is **NOT** in the target string

- **string::npos** is returned if no position exists

```
if (str.find("piano") == string::npos) {  
    do something here...    }  
    // This will happen if "piano" isn't in the string str
```

- You can search for a the **last occurrence** of a string in a string with the **.rfind** function

```
string str = "With a banjo on my knee and ban the bomb!";  
int rposition = str.rfind("ban");  
cout << rposition;    // Will display the number 28
```

# Search Functions 3

- `find_first_of`
  - Finds 1<sup>st</sup> occurrence of **any** of the characters included in the specified string
- `find_first_not_of`
  - Finds 1<sup>st</sup> occurrence of a character that is **not any** of the characters included in the specified string

- Example:

```
string card_number;  
cout << "Enter Credit Card Number: ";  
cin >> card_number;  
  
if (card_number.find_first_not_of("1234567890- ") != string::npos)  
{  
    cout << "The card number entered contains invalid characters"  
        << endl;  
}
```

# Descriptor Functions

- The **length** function returns the length of the string
  - The **size** function does the same thing...
  - So, if **string str1 = “Mama Mia!”**,  
then **str1.length() = 9**

Example – what will this code do?:

```
string name = “Bubba Smith”;  
for (int i = name.length(); i > 0; i--)  
    cout << name[i-1];
```



# Content Changers 1

## *append, erase*

- Use function **append** to append one string to another

```
string name1 = " Max";  
string name2 = " Powers";  
cout << name1.append(name2);    // Displays " Max Powers"
```

  - Does the same thing as: **name1 + name2**
  - Appends to the string and is a call by reference (**i.e. the string changes**)
- Use function **erase** to clear a string to an empty string
  - One use is: **name1.erase()** -- Does the same thing as: **name1 = ""**
  - Another use is: **name1.erase(start position, how many chars to erase)**
    - Erases part of the string and is a call by reference (**i.e. the string changes**)
    - Example:

```
cout << name2.erase(2, 2); // Displays " Pers"
```

# Content Changers 2

## *replace, insert*

- Use function **replace** to replace part of a string with another
  - Popular Usage:  
*string.replace(start position,  
places after start position to replace, replacement string)*
- Use function **insert** to insert a substring into a string
  - Popular Usage:  
*string.insert(start position, insertion string)*

### **Example:**

```
string country = "USA";  
cout << country.replace(2, 1, " of A"); // Displays "US of A"  
cout << country.insert(7, "BC");       // Displays "US of ABC"
```

# Content Changers 3

## *substr*

- Use function **substr** (short for “substring”) to extract and return a substring of the invoking **string** object
  - Popular Usage:  
*string.substr(start position, places after start position)*

### **Example:**

```
string city = “Santa Barbara”;  
cout << city.substr(3, 5)  
      // Displays “ta Ba”
```

# getline function

- For standard inputs, **cin** is fine
  - But it ignores space, tabs, and newlines
- Sometimes, you want to get the ***entire line of data from the input stream or file stream***
- Use the function **getline** for that purpose.
- It's from the **<istream>** library
  - **istream** is the “parent library” of ifstream
  - If you're already using <iostream> and <ifstream>, you **do not need** to include **<istream>**
  - istream is concerned with inputs from both keyboard and file streams
- Popular Usage:  

```
getline(ifstream, string);  
getline(cin, string);
```



# Program Example:

## getline demo

---

**DEMO!**

# TO DOs

---

- Homework #10 due Tuesday 11/1
- Lab #6
  - Due Friday, 11/4, at noon

**</LECTURE>**