COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 11: FILE I/O, FORMATTING, AND OTHER FILE OPERATIONS

SCHINNEL SMALL

USF
UNIVERSITY OF SOUTH FLORIDA

FILE I/O BASICS

FILE I/O BASICS

- File input/output allows us to use files instead of keyboard and monitor screen for program input and output respectively

- There are 5 steps to achieve basic file i/o:
  - obtain file stream header file
  - create objects
  - open file
  - read/write to file
  - close file

FILE I/O BASICS

- Step 1: We use the `fstream` header file for file access

- File stream types:
  - `ifstream` for input from a file
  - `ofstream` for output to a file
  - `fstream` for input from or output to a file

- Syntax: `#include <fstream>`

FILE I/O BASICS

- Step 2: Define file stream objects:

```
ifstream infile;
ofstream outfile;
```

- Step 3: Open file – create a link between file name (outside the program) and file stream object (inside the program)

- Use the `open` member function to read the file

```
infile.open("inventory.dat");
outfile.open("report.txt");
```

FILE I/O BASICS

- Step 3 (cont'd): Filename may include drive, path info
  - Suppose path is C:\data\inventory.txt
  - Syntax is: `inputFile.open("C:\\data\\inventory.txt");`
    (Note use of escape sequence)

- Input file must exist for `open` to work

- Output file will be created if necessary; existing file will be erased first

FILE I/O BASICS

- Tip: We can test a file stream object to detect if an open operation failed:

```
infile.open("test.txt");
if (!infile)
    cout << "File open failure!";
```

- Alternately, we can use the fail member function:

```
if (inputFile.fail())
    cout << "Error opening file.\n";
else
    { // Process the file. }
```

FILE I/O BASICS

- Step 4: Use >> and << with file objects to read/ write to files

- To read data from file to variables:
  - `infile >> partNum;`

- To write to file:
  - `outfile << "Inventory report";`

FILE I/O BASICS

- Tip: >> returns `true` when a value was successfully read, `false` otherwise

- Can be tested in a `while` loop to continue execution as long as values are read from the file:

```
while (inputFile >> number) ...
```

FILE I/O BASICS

- Step 5: Close the files

```
infile.close();
outfile.close();
```

- Don’t wait for operating system to close files at program end:
  - may be limit on number of open files
  - may be buffered output data waiting to send to file

FILE I/O EXAMPLES

- See Basic File I_O document on Canvas
  - Example 1: Write to file
  - Example 2: Read from file
  - Example 3: Read from file (with loop)

FILE I/O BASICS

- In many cases, you will want the user to specify the name of a file for the program to open

- In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function

- See Basic File I_O document (Example 4)

FORMATTING OUTPUT

FORMATTING OUTPUT

- Recall: In previous examples we used the iomanip header file to format our output
  - e.g. In week 2 (slide 53) we used `setw` to set the field width of output

- Here, we discuss other formatting options that are available for output

- Note: while examples may use objects from the `iostream` header file, they can also be applied to objects from `fstream` header file too

FORMATTING OUTPUT

- Recall: `setw(x)`: print in a field at least `x` spaces wide
  - Use more spaces if field is not wide enough

```
int num1 = 2897, num2 = 5,  num3 = 837;

// Display the first row of numbers
cout << setw(6) << num1 << setw(6)
     << num2 << setw(6) << num3 << endl;
```

Program Output

```
2897     5   837
```

- field width must be specified before each output

FORMATTING OUTPUT

- Other useful format options include:
  - `fixed`: use decimal notation for floating-point values
  - `scientific`: use scientific notation for floating point values
  - `setprecision(x)`:
    - when used with `fixed`, print floating-point value using `x` digits after the decimal point
    - Without `fixed`, print floating-point value using x significant digits
  - `showpoint:` always print decimal point for floating-point values

FORMATTING OUTPUT

- `setprecision` (without `fixed`) example

```
// This program demonstrates how setprecision rounds a
// floating point value.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
     double quotient, number1 = 132.364, number2 = 26.91;

     quotient = number1 / number2;
     cout << quotient << endl;
     cout << setprecision(5) << quotient << endl;
     cout << setprecision(4) << quotient << endl;
     cout << setprecision(3) << quotient << endl;
     cout << setprecision(2) << quotient << endl;
     cout << setprecision(1) << quotient << endl;
     return 0;
}
```

Program Output

```
4.91877
4.9188
4.919
4.92
4.9
5
```

FORMATTING OUTPUT

- `setprecision` (with `fixed`) example

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

int main()
{
    double quotient, number1 = 132.364, number2 = 26.91;

    quotient = number1 / number2;
    cout << fixed <<quotient << endl;
    cout << setprecision(5) << quotient << endl;
    cout << setprecision(4) << quotient << endl;
    cout << setprecision(3) << quotient << endl;
    cout << setprecision(2) << quotient << endl;
    cout << setprecision(1) << quotient << endl;
    return 0;
}
```

Output

```
4.918766
4.91877
4.9188
4.919
4.92
4.9
```

- Note that `fixed` is applied once (`scientific` used the same way)
  - `setprecision` can be applied only once if you expect the format to be the same for all subsequent output

FORMATTING OUTPUT

- Normally output is right justified, but you can confirm or change the alignment by using the `left` and `right` manipulators

```
double x = 146.789, y = 24.2, z = 1.783;
cout << left << setw(10) << x << endl;
cout << setw(10) << y << endl;
cout << setw(10) << z << endl;
```

- Note that the manipulator is only used once to affect the subsequent output

- It is often used with `setw` so make the effect more noticeable
  - Can also be used with `setfill` function to determine the type of character padding
  - e.g.
```
int x {10000};
    cout << left << setw(10) << setfill('^') << x;
```

FORMATTING OUTPUT – BASE VALUES

- C++ provides stream manipulators `dec`, `hex`, and `oct` to specify that integers would be displayed as decimal, hexadecimal and octal values

- In the stream extraction process, integers prefixed with `0` are treated as octal values, while integers prefixed with `0x` or `0X` are hexadecimal

- The stream manipulator `showbase` forces the base of an integer to be printed
  - `noshowbase` can be used to reverse the effect

FORMATTING OUTPUT – BASE VALUES

- C++ base example

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

int main()
{
    int number{65};
    cout << number << endl;
    cout << oct << number << endl;
    cout << hex << number << endl;
    cout << showbase << endl;
    cout << hex << number << endl;
    cout << oct << number << endl;
    cout << dec << number << endl;
}
```

Output

```
65
101
41

0x41
0101
65
```

FORMATTING OUTPUT – BOOLEAN

- Recall: boolean values are expressed in C++ as:
  - 1 – true
  - 0 – false

- Sometimes we wish to express the outcomes as strings
  - We can use the manipulator `boolalpha` for this
    - Used once; will affect all subsequent boolean output
  - To revert back to default, use `noboolaplha`

FORMATTING OUTPUT – BOOLEAN

- Example:

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

int main()
{
    bool value {true};
    cout << value << endl;
    cout << boolalpha << value << endl;
    value = false;
    cout << value << endl;
    cout << noboolalpha << value << endl;
}
```

Output

```
1
true
false
0
```

- Note that it does not require any additional header file

CHARACTERS AND STRING OBJECTS WITH INPUT

READING STRING INPUT

- Using `cin` with the >> operator to input strings can cause problems:

- It passes over and ignores any leading *whitespace characters (spaces, tabs, or line breaks)*

- To work around this problem, you can use the C++ function named `getline`

READING STRING INPUT

- Problem:

```
// This program illustrates a problem that can occur if
// cin is used to read character data into a string object.
#include <iostream>
#include <string>
using namespace std;

int main()
{
     string name;
     string city;

     cout << "Please enter your name: ";
     cin >> name;
     cout << "Enter the city you live in: ";
     cin >> city;
     cout << "Hello, " << name << endl;
     cout << "You live in " << city << endl;
     return 0;
}
```

Program Output with Example Input Shown in Bold

```
Please enter your name: Kate Smith [Enter]
Enter the city you live in: Hello, Kate
You live in Smith
```

READING STRING INPUT

- Solution:

```
// This program demonstrates using the getline function
// to read character data into a string object.
#include <iostream>
#include <string>
using namespace std;

int main()
{
     string name;
     string city;

     cout << "Please enter your name: ";
     getline(cin, name);
     cout << "Enter the city you live in: ";
     getline(cin, city);

     cout << "Hello, " << name << endl;
     cout << "You live in " << city << endl;
     return 0;
}
```

Program Output with Example Input Shown in Bold

```
Please enter your name: Kate Smith [Enter]
Enter the city you live in: Raleigh [Enter]
Hello, Kate Smith
You live in Raleigh
```

READING CHARACTER INPUT

- We've seen that we can read characters with cin
  - This was used when we overloaded the >> operator
- We can still encounter problems with reading whitespace characters
- To solve, we can use the get function (in various ways):
  - `cin.get()`
  - `char ch;  cin.get(ch);`
  - `ch = cin.get();`

MIXING CIN AND CIN.GET

- Mixing `cin >>` and `cin.get()` in the same program can cause input errors that are hard to detect
  - This can also occur when using `cin` and `getline`

- To skip over unneeded characters that are still in the keyboard buffer, use `cin.ignore()`:

```
cin.ignore(); // skip next char
cin.ignore(10, '\n'); // skip the next
                      // 10 char. or until a '\n'
```

MORE ON FILE OPERATIONS

FSTREAM DATA TYPE

- Recall: the fstream data type can be used to read from and write to files

- To distinguish between reading and writing, we use file access flags to specify the mode that is currently used
  - Sample modes:
    `ios::in` – input
    `ios::out` – output
  - Can be combined on `open` call:

```
dFile.open("class.txt", ios::in | ios::out);
```

FILE ACCESS FLAGS

File Access Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

- See Example 1 in the More File I_O Examples

PASSING FILE STREAM OBJECTS

- It is very useful to pass file stream objects to functions
- However, be sure to always pass file stream objects by reference
  - See Example 2 in More File I_O Examples
- You can use member functions like `get` to read a single character from a file, and `put` to write a character to a file
  - Examples:
```
char letterGrade;
gradeFile.get(letterGrade);
```
  - `reportFile.put(letterGrade);`