C++ Lecture 7

* File Input and Output, Output Formatting, and Character Processing
* CIS 251 • Shelby-Hoover Campus

File I/O Basics

* cin and cout are both **streams** (flows of characters into or out of a program)
* To use files for input and output, replace cin and cout with file stream variables
  + These **class** types are provided in the library <fstream>
  + The class type for input files is ifstream
  + The class type for output files is ofstream
  + Variables declared from these types are known as **objects**
* Some operations with streams are accomplished using **member functions**, also known as **methods**
  + The object used to execute the method is the **calling object**
  + The object and method names are joined by the **dot operator**
  + Example: objectVariable.methodName(arguments);

Setup and Opening

* First, declare the variable of the appropriate type (one for each file to be opened; you cannot use a single variable for multiple files at the same time):  
    
  ifstream inFileVariable;  
  ofstream outFileVariable;
* Next, invoke the method open with the file name:  
    
  inFileVariable.open("InFileName.txt");  
  outFileVariable.open("OutFileName.txt");  
  + Visual Studio looks for input files, and creates output files, in the project folder unless another path is specified in the quotation marks
  + If an output file already exists, it is overwritten (the original contents are deleted) unless a flag (provided in the library <iostream>) is added to the open method call indicating to **append** to the file instead:  
      
    outFileVariable.open("OutFileName.txt", ios::app);

I/O Statements and Closing

* The input and output statements for a file are the same as for console input and output, with the stream variable names replacing cin and cout:  
    
  inFileVariable >> value1 >> value2;  
  outFileVariable << "value1 is " << value1 << endl;
* The direction of the stream insertion operators should be consistent with the operation (input vs. output)
* Once the operations using a particular file are complete, close the file to force any buffered output statements to be written to the file and to prevent damage to the files:  
    
  inFileVariable.close();  
  outFileVariable.close();

File I/O Example

* Copying three int values from one file to another, with spaces between the values:  
    
  #include <fstream>  
  using namespace std;  
    
  int main()  
  {  
   ifstream originalFile;  
   ofstream newFile;  
   int val1, val2, val3;  
    
   originalFile.open("Original.txt");  
   newFile.open("New.txt");  
    
   originalFile >> val1 >> val2 >> val3;  
   newFile << val1 << " " << val2 << " " << val3 << endl;  
    
   originalFile.close();  
   newFile.close();  
    
   return 0;  
  }

Other File I/O Details

* A program may need to detect if there is a problem opening a file
  + The method fail() can be used to see if the open method call was unsuccessful
  + If fail() returns true, display an error message and call the function exit (which may require #include <cstdlib>)
  + Example:  
      
    inFileVariable.open("MyInputFile.txt");  
    if (inFileVariable.fail())  
    {  
     cout << "Error opening input file.\n";  
     exit(1); // 1 indicates an error  
    }
* Don’t prompt the user for values coming from the file

User-Defined File Names

* A user may specify the name of the file to be opened for input or output
* The variable used to store this file name is a character array:  
    
  char inFileName[number\_of\_characters];  
  + Don’t use the same variable name as the stream variable
  + The number in the square brackets should be one more than the maximum number of characters expected for a special “end of string” character (covered in chapter 8)
  + After using the character array in an input statement, pass it to the call to open:  
      
    inFileVariable.open(inFileName);

Formatting Output

* Both cout and ofstream variable output can be formatted
* The method setf supports additional flags:
  + ios::fixed ensures that scientific notation is avoided
  + ios::scientific ensures that scientific notation is used
  + ios::showpoint requires the decimal on a floating-point value
  + ios::showpos displays a + before positive integer values
  + ios::right and ios::left specify alignment within fields
* Any of these formatting changes can be applied using setf and removed using unsetf:  
    
  outFileVariable.setf(ios::showpoint);  
  outFileVariable.unsetf(ios::right);

Precision and Width

* Set the number of digits after the decimal point using precision:  
    
  outFileVariable.precision(2);
* A program can also specify the field width for each output:  
    
  outFileVariable.width(10);  
  + If the next value is smaller than the width, spacing will be added to line up with the alignment (default is right)
  + If the next value is larger, it will overflow the width
  + Width must be set for each value individually

Manipulators

* Precision and width can also be applied inline in an output statement using **manipulator** functions
* These require the library <iomanip> and the standard namespace
* The manipulator for width is setw(width), and the manipulator for precision is setprecision(digits)
  + cout << setprecision(2) << salary << endl;
  + outFileVariable << setw(10) << firstName << endl;
  + setprecision is “sticky” (the program remembers this setting), but setw must be called each time it is needed

Reading an Entire File

* A programmer may not know how many values an input file contains before the program uses it
* An input statement returns a bool value indicating whether the program was successful in obtaining the value from the file
  + Once the program reaches the end of the file, the statement returns a value of false
  + Using the input statement as the Boolean expression for a while loop allows the program to read and process every value in the file
* A programmer may also invoke the method eof() after an input statement to determine whether the program has reached the end of the input file
  + The program may loop while this function returns false (using the ! operator)
  + This requires an input statement before the loop header and another input statement as the last statement in the loop body (essentially, this is a sentinel-controlled loop)

Input Loop Example 1

* Using the input statement as a Boolean expression controlling a loop that calculates the total of the values in an input file:  
    
  int someNum, sum = 0;  
  ifstream inFile;  
    
  inFile.open("LotsOfStuff.txt");  
    
  while (inFile >> someNum)  
   sum += someNum;  
    
  inFile.close();  
    
  cout << "The sum of the input file values is " << sum << endl;

Input Loop Example 2

* Using the value returned by the function eof() as a Boolean expression controlling a loop that calculates the total of the values in an input file (notice the separate input statements):  
    
  int someNum, sum = 0;  
  ifstream inFile;  
    
  inFile.open("LotsOfStuff.txt");  
    
  inFile >> someNum; // priming input  
  while (!inFile.eof())  
  {  
   sum += someNum;  
   inFile >> someNum; // input at end of loop body  
  }  
    
  inFile.close();   
    
  cout << "The sum of the input file values is " << sum << endl;

File I/O and Functions

* Remember that a variable is local to the function in which it is declared; this includes stream variables
* If a file is to be used in multiple functions, the stream variable associated with it should be passed as a reference parameter (with an & after the parameter type) from the first function to use the file to any other function using the file
* The first function that uses the file should open it, and the last function to use the file should close it
* Example Prototype:  
  void make\_neat(ifstream& oldFile, ofstream& newFile, int digits,  
   int characters);

File I/O Summary

* The key to reading from and writing to files rather than the console is to declare file stream variables that are used to open the files
  + ifstream: input file stream data type
  + ofstream: output file stream data type
* Anything that can be done with cin and cout can also be done with ifstream and ofstream variables
  + Plug the variable name in place of cin or cout
  + Be careful: if a program uses cin or cout when it should be handling a file, it won’t generate the desired result

Character Input

* When a program uses normal input statements with >> operators, it skips whitespace (spaces, tabs, new lines) to get to the next item to be read
* A program can read one character at a time, including the whitespace characters
* There’s a special method, .get(charVarName), that reads the next character from the input stream and stores it in the char variable argument
  + cin.get(nextCh);
  + myInputFile.get(oneCh);

Character Input Example

* Echoing the characters to the end of the line (section 6.3):  
    
  char symbol;  
    
  cout << "Enter a line of input and I will echo it:" << endl;  
    
  do  
  {  
   cin.get(symbol);  
   cout << symbol;  
  } while (symbol != '\n');  
    
  cout << "That's all, folks!" << endl;

Other Character I/O

* Just as get captures a character from any input stream, put sends a single character to any output stream
  + cout.put(letterGrade);
  + myOutputFile.put(maritalStatus);
* If a single character that has been read needs to be read again later, it can be sent back to the input stream using the putback method
  + cin.putback(lastCh);
  + myInputFile.putback(readCh);
  + Don’t use putback with an output stream!

Handling Unwanted Input

* There’s no guarantee that a line of input will contain exactly what the program expects
* A program can use the get method to clear out any extra input remaining in the current line (Display 6.7):  
    
  void new\_line( )  
  {  
   char symbol;  
   do  
   {  
   cin.get(symbol);  
   } while (symbol != '\n');  
  }

Stream Parameters

* The function in the previous example assumes that the input is captured via the console – it won’t work for file input
* If a function has a parameter of type ifstream& or ofstream&, the argument must be a file stream – it cannot be cin or cout
* C++ provides parameter types that work for both console and file streams
  + Input: istream&
  + Output: ostream&

Stream Parameter Example

* Rewriting new\_line with a stream parameter:  
    
  void new\_line(istream& in\_stream)  
  {  
   char symbol;  
   do  
   {  
   in\_stream.get(symbol);  
   } while (symbol != '\n');  
  }
* The argument to new\_line should be either cin or an ifstream variable

Default Arguments

* When a function header contains parameters, a program usually must provide an argument for each parameter
* It’s possible to write a function definition in such a way that it works properly even when there are fewer arguments than there are parameters
* If the header of a function definition includes **default arguments** for some of the parameters, the program will use these values when an argument is not specified for those parameters
  + Providing a default argument for a parameter is similar to initializing a variable when it is declared: add the assignment operator and default argument value after the parameter name
  + You cannot skip a parameter with default arguments; if you provide a default argument for a parameter in the middle of the parameter list, all parameters after that parameter must also have default arguments
* You could rewrite new\_line to assume cin if there’s no argument:  
  void new\_line(istream& in\_stream = cin)

Default Arguments Example

* Function definition with default arguments:  
  void default\_args(int par1, int par2, int par3 = -3, int par4 = -4)  
  {  
   cout << par1 << ' ' << par2 << ' ' << par3 << ' ' << par4 << endl;  
  }
* Various function calls that make use of the default arguments:  
  default\_args(5, 6); // 5 6 -3 -4  
  default\_args(6, 7, 8); // 6 7 8 -4  
  default\_args(5, 6, 7, 8); // 5 6 7 8

The ignore Method

* Another way of clearing unwanted characters is to use the ignore method
  + The first argument is the number of characters to skip
  + The second argument is a terminal character to skip past
  + The method will clear any characters out of the stream that meet the first of the two criteria reached
  + Example: cin.ignore(80, '\n');
* Be careful when mixing the input stream insertion operator and the get method
  + If you use get after a regular input statement, whatever whitespace character exists after the first input (very often '\n') will be placed in the character argument to get
  + Clear the whitespace before adding a get statement

Character Processing

* The library cctype provides a set of functions to examine and convert characters of various forms
* The argument to each function must be of type char
* Display 6.9 is a table of some of these character functions
  + The functions that start with to return an int containing the ASCII code for a particular character; these can be assigned to char variables, or they can be cast as char values in output statements
  + The functions that start with is return true or false; these can be used as Boolean expressions in decision or loop structures, or their return values can be assigned to bool variables

cctype Conversions

* toupper(char\_exp)
  + If char\_exp is a lowercase letter, toupper returns the ASCII value of the uppercase equivalent of char\_exp
  + If not, toupper returns the ASCII value of char\_exp itself
* tolower(char\_exp)
  + If char\_exp is an uppercase letter, tolower returns the ASCII value of the lowercase equivalent of char\_exp
  + If not, tolower returns the ASCII value of char\_exp itself

cctype Evaluations

|  |  |
| --- | --- |
| **Function** | **Returns true if char\_exp is:** |
| isupper(char\_exp) | An uppercase letter |
| islower(char\_exp) | A lowercase letter |
| isalpha(char\_exp) | A letter (regardless of case) |
| isdigit(char\_exp) | A digit (e.g., '0') |
| isspace(char\_exp) | Any whitespace character (' ', '\t', '\n') |