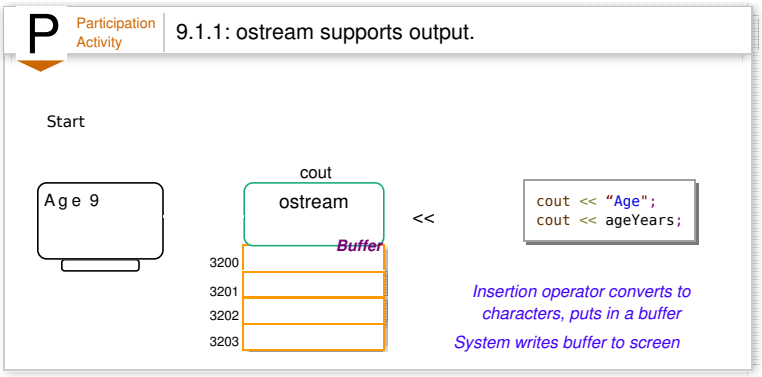
Chapter 9 - Streams

Section 9.1 - The ostream and cout streams

Programs need a way to output data to a screen, file, or elsewhere. An ***ostream***, short for "output stream," is a class that supports output, available via `#include <iostream>` and in namespace "std". ostream provides the << operator, known as the insertion operator, for converting different types of data into a sequence of characters. That sequence is normally placed into a buffer, and the system then outputs the buffer at various times.

cout is a predefined ostream object (e.g., you can think of it as defined as `ostream cout;` in the iostream library) that is pre-associated with a system's standard output, usually a computer screen. The following animation illustrates.



The << operator is overloaded with functions to support the various standard data types, such as int, bool, float, etc., each function converting that data type to a sequence of characters. The operator may be further overloaded by the string library from `#include <string>` or by the programmer for programmer-created classes.

The << operator returns a reference to the ostream that called it, and is evaluated from left to right like most operators, so << operators can appear in series.

Figure 9.1.1: Insertion operator.

```
cout << "Num" << myInt;
```

can be thought of as:

```
( cout.operator<<("Num") ).operator<<(myInt);
```

Basic use of cout and the insertion operator were covered in an earlier section.

Participation Activity 9.1.2: ostream and cout.

#	Question	Your answer
1	Characters written to cout are immediately written to a system's standard output.	True
		False
2	To use cout, a program must include the statement `#include <iostream>`.	True
		False
3	<< is known as the stream operator.	True
		False

Exploring further:

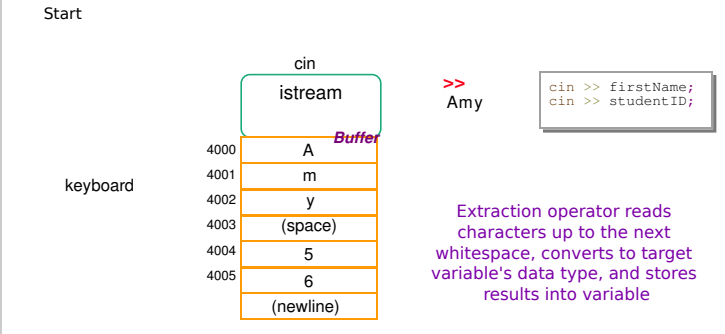
- Ostream Reference Page from cplusplus.com
- More on Ostreams from msdn.microsoft.com

Section 9.2 - The istream and cin streams

Programs need a way to receive input data, from a keyboard, touchscreen, or elsewhere. An ***istream***, short for "input stream," is a class that supports input. Available via `#include <iostream>`, istream provides the >> operator, known as the ***extraction operator***, to extract data from a data buffer and to write the data into different types of variables.

cin is a predefined istream (you can think of it as defined as `istream cin;` in the iostream library) that is pre-associated with a system's standard input, which is usually a computer keyboard. The system automatically puts the standard input into a data buffer associated with cin, from which >> can extract data. The following animation illustrates.

Participation Activity 9.2.1: istream supports input.



Basic use of cin and the extraction operator were covered in an earlier section.

Participation Activity 9.2.2: istream and cin.

#	Question	Your answer
1	cin is a predefined istream associated with the system's standard input.	True
		False
2	To use cin, a program must include the statement `#include <istream>`.	True
		False
3	A read from cin will directly read characters from the system's keyboard.	True
		False
4	>> is known as the extraction operator.	True
		False

Exploring further:

- istream Reference Page from cplusplus.com
- More on istreams from msdn.microsoft.com

Section 9.3 - Output formatting

A programmer can adjust the way that output appears, a task known as output formatting. The main formatting approach uses manipulators. A ***manipulator*** is an item designed to be used with the insertion operator << or extraction operator >> to adjust the way output appears, and is available via `#include <iomanip>;` or `#include <ios>;` in namespace std. For example, `cout << setprecision(3) << myFloat;` causes the floating-point variable myFloat to be output with only 3 digits; if myFloat was 12.34, the output would be 12.3.

Most manipulators change the state of the stream such that the manipulation affects all subsequent output, not just the next output.

Manipulating floating-point output is commonly done. For the following, assume a sample value of 12.34.

Table 9.3.1: Floating point manipulators.

Manipulator	Description	Example
fixed	Use fixed-point notation. From <ios>	12.34
scientific	Use scientific notation. From <ios>	1.234e+01
setprecision(p)	If stream has not been manipulated to fixed or scientific: Sets max number of digits in number	p=3 yields 12.3 p=5 yields 12.34
	If stream has been manipulated to fixed or scientific: Sets max number of digits in fraction only (after the decimal point). From <iomanip>	fixed: p=1 yields 12.3 scientific: p=1 yields 1.2e+01
showpoint	Even if fraction is 0, show decimal point and trailing 0s. Opposite is noshowpoint. From <ios>	For 99.0 with precision=2 and fixed: 99 (default or noshowpoint) 99.00 (showpoint)

Figure 9.3.1: Example output formatting for floating-point numbers.

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

int main() {

   double milesTrvld = 765.4321;

   cout << "setprecision(p) -- Sets # digits" << endl;
   cout << milesTrvld << " (default p is 6)" << endl;
   cout << setprecision(8) << milesTrvld << " (p = 8)" << endl;
   cout << setprecision(5) << milesTrvld << " (p = 5)" << endl;
   cout << setprecision(2) << milesTrvld << " (p = 2)"
        << " (note rounding)" << endl;
   cout << milesTrvld << " (manipulator persists)" << endl << endl;

   cout << setprecision(2);
   cout << "(For following, p = 2 applies to fraction only)" << endl;

   // fixed -- uses fixed point notation
   cout << fixed;
   cout << "fixed: " << milesTrvld << endl;

   // scientific -- uses scientific notation
   cout << scientific;
   cout << "scientific: " << milesTrvld << endl;

   return 0;
}
```

```
setprecision(p) -- Sets # digits
765.432 (default p is 6)
765.4321 (p = 8)
765.43 (p = 5)
7.7e+02 (p = 2) (note rounding)
7.7e+02 (manipulator persists)

(For following, p = 2 applies to fraction only)
fixed: 765.43
scientific: 7.65e+02
```

Participation Activity

9.3.1: Output formatting for floating-point manipulators.

For each question, assume no manipulators have previously been applied. Given the following,

```
double temp = 98.63;
// a floating-point manipulator
cout << temp;
```

#	Question	Your answer
1	Which would cause the output to appear as "98.6"?	`cout << fixed;`
		`cout << setprecision(3);`
		`cout << setprecision(2);`
		`cout << scientific << setprecision(2);`
2	Which would cause the output to appear as "9.86e+01"?	`cout << fixed;`
		`cout << setprecision(3);`
		`cout << setprecision(2);`
		`cout << scientific << setprecision(2);`
3	Which would cause the output to appear as "99"?	`cout << fixed;`
		`cout << setprecision(3);`
		`cout << setprecision(2);`
		`cout << scientific << setprecision(2);`

Table 9.3.2: Certain text manipulators help align output.

Manipulator	Description	Example (for item "Amy")
setw(n)	Sets the number of characters for the next output item only (does not persist, in contrast to other manipulators). By default, the item will be right-aligned, and filled with spaces. From <iomanip>	For n=7: " Amy"
setfill(c)	Sets the fill to character c. From <iomanip>	For c='*': "****Amy"
left	Changes to left alignment. From <ios>	"Amy "
right	Changes back to right alignment. From <ios>	" Amy"

Figure 9.3.2: Example illustrating manipulators useful for output alignment.

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

int main() {
   cout << "Dog age in human years (dogyears.com)" << endl << endl;

   // set num char for each column, set alignment
   cout << setw(10) << left  << "Dog age" << "|";
   cout << setw(12) << right << "Human age" << endl;
   cout << "-----------------------------" << endl;
   cout << setw(10) << left  << "2 months" << "|";
   cout << setw(12) << right << "14 months" << endl;
   cout << setw(10) << left  << "6 months" << "|";
   cout << setw(12) << right << "5 years" << endl;

   // set fill character, num char for each column, set alignment
   cout << setfill('-');
   cout << setw(10) << left  << "8 months" << "|";
   cout << setw(12) << right << "9 years" << endl;
   cout << setw(10) << left  << "1 year" << "|";

   // change fill character, num char for each column, set alignment
   cout << setfill('.');
   cout << setw(12) << right << "15 years" << endl;

   // change fill character, num char for each column
   cout << setfill('*') << setw(30) << "" << endl;

   return 0;
}
```

```
Dog age in human years (dogyears.com)

Dog age   |   Human age
------------------------------
2 months  |   14 months
6 months  |     5 years
8 months--|-----9 years
1 year----|....15 years
******************************
```

Of particular interest is how the setw() and setfill() manipulators are used in the last few lines. Note how they are used to create a line of 30 asterisks, without having to type 30 asterisks.

Most manipulators are persistent, meaning they change the state of the stream for all subsequent output. The exception is setw(), which only affects the next output item, defined that way likely because programmers usually only want to set the width of the next item and not all subsequent items.

Some additional manipulators are:

Table 9.3.3: Additional manipulators.

Manipulator	Description
endl	Inserts a newline character '\n' into the output buffer, and informs the system to flush the buffer. From <iostream>
flush	Informs the system to flush the buffer. From <iostream>

Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources; once those resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print. As such, the system may wait until the buffer is full, or at least has a certain number of characters, before moving them to the output device. Or, with fewer characters in the buffer, the system may wait until the resources are not busy. However, sometimes a programmer does not want the system to wait. For example, in a very processor-intensive program, such waiting could cause delayed and/or jittery output. If the desired output is a line, the programmer may use endl. If the desired output is just text within a line, the programmer can use flush.

A common error is to assume that a cout statement is never reached because the output had not been flushed when the program crashed.

Manipulators are actually functions. The reason they are functions is to make them work with the overloaded << and >> operators, but details are beyond our scope. However, of relevance is that a common error is to have a statement like `setprecision(2);` rather than `cout << setprecision(2);`, which compiles fine but does not impact cout.

More manipulators exist. See cplusplus.com

Participation Activity

9.3.2: Output formatting for text manipulators.

For each question, assume no manipulators have previously been applied. Given the following,

```
string str = "Amy";
```

#	Question	Your answer
1	Which statement will print "...Amy"?	`cout << setfill('.') << str;`
		`cout << setw(6) << setfill('.') << str;`
		`cout << setw(6) << "." << str;`
		`cout << right << setw(6) << str;`
2	Which statement will print "Amy"?	`cout << setfill('.') << str;`
		`cout << setw(6) << setfill('.') << str;`
		`cout << setw(6) << "." << str;`
		`cout << right << setw(6) << str;`
3	Which statement will print " Amy"?	`cout << setfill('.') << str;`
		`cout << setw(6) << setfill('.') << str;`
		`cout << setw(6) << "." << str;`
		`cout << right << setw(6) << str;`

Challenge Activity

9.3.1: Output formatting: Printing a maximum number of digits.

Write a single statement that prints outsideTemperature with 4 digits. End with newline. Sample output:

```
103.5
```

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

int main() {
   double outsideTemperature = 103.45632;

   /* Your solution goes here  */

   return 0;
}
```

Run

Challenge Activity

9.3.2: Output formatting: Printing a maximum number of digits in the fraction.

Write a single statement that prints outsideTemperature with 2 digits in the fraction (after the decimal point). End with a newline. Sample output:

```
103.46
```

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

int main() {
   double outsideTemperature = 103.45632;

   /* Your solution goes here  */

   return 0;
}
```

Run

Section 9.4 - String streams

Sometimes a programmer wishes to read input data from a string rather than from the keyboard (standard input). A new input string stream variable of type ***istringstream*** can be created that is associated with a string rather than with the keyboard (standard input). istringstream is derived from istream. Such a stream can be used just like the cin stream. The following program illustrates.

Figure 9.4.1: Reading a string as an input stream.

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

int main() {
   string  userInfo = "Amy Smith 19"; // Input string
   istringstream inSS(userInfo);      // Input string stream
   string firstName;                  // First name
   string lastName;                   // Last name
   int userAge = 0;                   // Age

   // Parse name and age values from input string
   inSS >> firstName;
   inSS >> lastName;
   inSS >> userAge;

   // Output parsed values
   cout << "First name: " << firstName << endl;
   cout << "Last  name: " << lastName << endl;
   cout << "Age: "        << userAge << endl;

   return 0;
}
```

```
First name: Amy
Last  name: Smith
Age: 19
```

The program uses `#include <sstream>` for access to the string stream class, which is in namespace std. The line `istringstream inSS(userInfo);` defines a new stream variable and initializes its buffer to a copy of userInfo. Then, the program can extract data from stream inSS using >> similar to extracting from cin.

A common use of string streams is to process user input line-by-line. The following program reads in the line as a string, and then extracts individual data items from that string.

Figure 9.4.2: Using a string stream to process a line of input text.

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

int main() {
   istringstream inSS;       // Input string stream
   string lineString;        // Holds line of text
   string firstName;         // First name
   string lastName;          // Last name
   int    userAge = 0;       // Age
   bool   inputDone = false; // Flag to indicate next iteration

   // Prompt user for input
   cout << "Enter \"firstname lastname age\" on each line" << endl;
   cout << "(\"Exit\" as firstname exits)." << endl << endl;

   // Grab data as long as "Exit" is not entered
   while (!inputDone) {

      // Entire line into lineString
      getline(cin, lineString);

      // Copies to inSS's string buffer
      inSS.clear();
      inSS.str(lineString);

      // Now process the line
      inSS >> firstName;

      // Output parsed values
      if (firstName == "Exit") {
         cout << "   Exiting." << endl;

         inputDone = true;
      }
      else {
         inSS >> lastName;
         inSS >> userAge;

         cout << "   First name: " << firstName << endl;
         cout << "   Last  name: " << lastName << endl;
         cout << "   Age:        " << userAge   << endl;
         cout << endl;
      }
   }

   return 0;
}
```

```
Enter "firstname lastname age" on each line
("Exit" as firstname exits).

Mary Jones 22
   First name: Mary
   Last  name: Jones
   Age:        22

Sally Smith 14
   First name: Sally
   Last  name: Smith
   Age:        14

Exit
   Exiting.
```

The program uses getline to read an input line into a string. The line `inSS.str(lineString);` uses the str(s) function to initialize the stream's buffer to string s. Afterwards, the program extracts input from that stream using >>. The statement `inSS.clear();` is necessary to reset the state of the stream so that subsequent extractions start from the beginning; the clear resets the stream's state.

Similarly, a new output string stream variable of type ***ostringstream*** can be created that is associated with a string rather than with the screen (standard output). ostringstream is a special kind of (i.e., is derived from) ostream. Once defined, a program can insert characters into that stream using <<, as follows.

Figure 9.4.3: Output string stream example.

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

int main() {
   ostringstream fullNameOSS; // Output string stream
   ostringstream ageOSS;      // Output string stream
   string firstName;          // First name
   string lastName;           // Last name
   string fullName;           // Full name (first and last)
   string ageStr;             // Age (string)
   int userAge = 0;           // Age

   // Prompt user for input
   cout << "Enter \"firstname lastname age\": " << endl;
   cin >> firstName;
   cin >> lastName;
   cin >> userAge;

   // Writes to buffer, then copies from buffer into string
   fullNameOSS << lastName << ", " << firstName;
   fullName = fullNameOSS.str();

   // Output parsed input
   cout << endl << "   Full name: " << fullName << endl;

   // Writes int age as chars to buffer
   ageOSS << userAge;

   // Appends (minor) to buffer if less than 21, then
   // copies buffer into string
   if (userAge < 21) {
      ageOSS << " (minor)";
   }

   ageStr = ageOSS.str();

   // Output string
   cout << "   Age: " << ageStr << endl;

   return 0;
}
```

```
Enter "firstname lastname age":
Mary Jones 22

   Full name: Jones, Mary
   Age: 22

...

Enter "firstname lastname age":
Sally Smith 14

   Full name: Smith, Sally
   Age: 14 (minor)
```

After defining an output string stream and inserting characters into its buffer, the program uses the ***str()*** function to copy that buffer to a string variable. Note that the str() function here has no parameters, in contrast to the example used for istringstream above.

Participation Activity 9.4.1: Input/output string streams.

#	Question	Your answer
1	Define an istringstream variable named inSS that creates an input string stream using the String variable myStrLine.	
2	Write a statement that sets the buffer of the input string stream called inSS to a different string called lineString.	
3	Given an ostringstream variable called outSS, write a statement that appends the value of the int variable carSpeed to the stream's buffer.	
4	Write a statement that copies the contents of an output string stream to a string variable called myStr. Assume the ostringstream variable is called outSS.	

Exploring further:

- stringstream Reference Page from cplusplus.com

Challenge Activity

9.4.1: Reading from a string.

Write code that uses the input string stream inSS to read input data from string userInput, and updates variables userMonth, userDate, and userYear. Sample output if userinput is "Jan 12 1992":

```
Month: Jan
Date: 12
Year: 1992
```

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

int main() {
   string userInput = "Jan 12 1992";
   istringstream inSS(userInput);
   string userMonth;
   int userDate = 0;
   int userYear = 0;

   /* Your solution goes here  */

   cout << "Month: " << userMonth << endl
        << "Date: " << userDate << endl
        << "Year: " << userYear << endl;

   return 0;
}
```

Run

Challenge Activity

9.4.2: Output using string stream.

Write code that inserts userItems into the output string stream itemsOSS until the user enters "Exit". Each item should be followed by a space. Sample output if user input is "red purple yellow Exit":

```
red purple yellow
```

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

int main() {
   string userItem;
   ostringstream itemsOSS;

   cout << "Enter items (type Exit to quit):" << endl;
   cin >> userItem;

   while (userItem != "Exit") {

      /* Your solution goes here  */

      cin >> userItem;
   }

   cout << itemsOSS.str() << endl;

   return 0;
}
```

Run

Section 9.5 - File input/output

Sometimes a program should get input from a file rather than from a user typing on a keyboard. To achieve this, a programmer can create a new input stream that comes from a file, rather than the predefined input stream cin that comes from the standard input (keyboard). That new input stream can then be used just like cin, as the following program illustrates. Assume a text file exists named myfile.txt with the contents shown (created for example using Notepad on a Windows computer or using TextEdit on a Mac computer).

Figure 9.5.1: Input from a file.

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

int main() {
   ifstream inFS;     // Input file stream
   int fileNum1 = 0;  // Data value from file
   int fileNum2 = 0;  // Data value from file

   // Try to open file
   cout << "Opening file myfile.txt." << endl;

   inFS.open("myfile.txt");
   if (!inFS.is_open()) {
      cout << "Could not open file myfile.txt." << endl;
      return 1; // 1 indicates error
   }

   // Can now use inFS stream like cin stream
   // myfile.txt should contain two integers, else problems
   cout << "Reading two integers." << endl;
   inFS >> fileNum1;
   inFS >> fileNum2;
   cout << "Closing file myfile.txt." << endl;
   inFS.close(); // Done with file, so close it

   // Ouput values read from file
   cout << "num1: " << fileNum1 << endl;
   cout << "num2: " << fileNum2 << endl;
   cout << "num1 + num2: " << (fileNum1 + fileNum2) << endl;

   return 0;
}
```

myfile.txt with two integers:

```
5
10
```

```
Opening file myfile.txt.
Reading two integers.
Closing file myfile.txt.
num1: 5
num2: 10
num1 + num2: 15
```

Five lines are needed for the new input stream, highlighted above.

- The `#include <fstream>` (for "file stream") enables use of the file stream class.
- A new stream variable has been defined: `ifstream inFS;`. ifstream is short for input file stream, and is derived from istream.
- The line `inFS.open("myfile.txt");` opens the file for reading and associates the file with the inFS stream. Because of the high likelihood that the open fails, usually because the file does not exist or is in use by another program, the program checks whether the open was successful using `if (!inFS.is_open())`.
- The successfully opened input stream can then be used just like the cin stream, e.g., using `inFS >> num1;` to read an integer into num1.
- Finally, when done using the stream, the program closes the file using `inFS.close()`.

A common error is to type `cin >> num1;` when actually intending to get data from a file as in `inFS >> num1`. Another common error is a mismatch between the variable data type and the file data, e.g., if the data type is int but the file data is "Hello".

Try 9.5.1: Good and bad file data.

File input, with good and bad data: Create myfile.txt with contents 5 and 10, and run the above program. Then, change "10" to "Hello" and run again, observing the incorrect output.

The `inFS.open(str)` function has a string parameter str that is a C string, meaning a null-terminated char array, such as a string literal like "myfile.txt". Unlike many other functions with a string parameter, the function has not been overloaded to alternatively accept a C++ string. Thus, the following code is an error.

Figure 9.5.2: The file.open() function requires a C string argument; a C++ string yields a (bewildering) compiler error message.

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

int main() {
   ifstream inFS;                 // Input file stream
   string filename = "myfile.txt"; // Input file name

   // Try to open file
   inFS.open(filename);

   // rest of program...
}
```

```
$ g++ -Wall fileopentest.cpp
fileopentest.cpp: In function 'int main()':
fileopentest.cpp:9: error: no matching function for call to
'std::basic_ifstream >::open(std::string&)'
/usr/include/c++/4.2.1/fstream:518: note: candidates are:
void std::basic_ifstream<_CharT, _Traits>::open(const
char*, std::_Ios_Openmode)
[with _CharT = char, _Traits = std::char_traits]
```

Programmers commonly want to use a C++ string, e.g., after having the user input the filename via `cin >> filename;`. The solution is to use the function c_str() that comes with C++ strings. If filename is a C++ string, then filename.c_str() returns a C string.

Figure 9.5.3: Using c_str() to convert a C++ string to a C string before passing the string to the file open function.

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

int main() {
   ifstream inFS;                 // Input file stream
   string filename = "myfile.txt"; // Input file name

   // Try to open file
   cin >> filename;
   inFS.open(filename.c_str());

   // rest of program...
}
```

The following provides another example wherein the program reads items into three int variables. For this program, myfile.txt must have 3 integers, e.g., 5 10 20.

Figure 9.5.4: Program that reads data from myfile.txt into a three int variables.

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

int main() {
   ifstream inFS;     // Input file stream
   int fileNum1 = 0;  // File data
   int fileNum2 = 0;  // File data
   int fileNum3 = 0;  // File data

   // Open file
   inFS.open("myfile.txt");

   if (!inFS.is_open()) {
      cout << "Could not open file myfile.txt." << endl;

      return 1;
   }

   // Get numbers. If too few, may encounter problems
   inFS >> fileNum1;
   inFS >> fileNum2;
   inFS >> fileNum3;

   // Done with file, close it
   inFS.close();

   // Print numbers read from file
   cout << "Numbers: ";
   cout << fileNum1 << " ";
   cout << fileNum2 << " ";
   cout << fileNum3 << " ";
   cout << endl;

   return 0;
}
```

myfile.txt with two integers:

```
5
10
20
```

```
Numbers: 5 10 20
```

A program can read varying amounts of data in a file by using a loop that reads until the end of the file has been reached, as follows.

Figure 9.5.5: Reading a varying amount of data from a file.

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

int main() {
   ifstream inFS;   // Input file stream
   int fileNum = 0; // File data

   // Open file
   cout << "Opening file myfile.txt." << endl;
   inFS.open("myfile.txt");

   if (!inFS.is_open()) {
      cout << "Could not open file myfile.txt." << endl;

      return 1;
   }

   // Print read numbers to output
   cout << "Reading and printing numbers." << endl;

   inFS >> fileNum;
   while (!inFS.eof()) {
      cout << "num: " << fileNum << endl;

      inFS >> fileNum;
   }
   cout << "num: " << fileNum << endl;

   cout << "Closing file myfile.txt." << endl;

   // Done with file, so close it
   inFS.close();

   return 0;
}
```

myfile.txt with variable number of integers:

```
111
222
333
444
555
```

```
Opening file myfile.txt.
Reading and printing numbers.
num: 111
num: 222
num: 333
num: 444
num: 555
Closing file myfile.txt.
```

The ***eof()*** function returns true if the previous stream operation reached the end of the file.

The function ***good()*** is sometimes used instead of eof() due to being more general, evaluating to true if the previous stream operation had no problem, where a problem can include end-of-file, corrupt data, etc. In that case, the loop statement would be `while (inFS.good()) {...}`.

Similarly, a program may write output to a file rather than to standard output, as shown below. The program defines a variable of type ***ofstream***, which is a kind of (i.e., is derived from) ostream.

Figure 9.5.6: Sample code for writing to a file.

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

int main() {
   ofstream outFS; // Output file stream

   // Open file
   outFS.open("myoutfile.txt");

   if (!outFS.is_open()) {
      cout << "Could not open file myoutfile.txt." << endl;
      return 1;
   }

   // Write to file
   outFS << "Hello" << endl;
   outFS << "1 2 3" << endl;

   // Done with file, so close it
   outFS.close();

   return 0;
}
```

Contents of myoutfile.txt after running the program:

```
Hello
1 2 3
```

Participation Activity 9.5.1: File input/output.

#	Question	Your answer

1	What is the error in the following code?
	`ifstream inFS;`
	`int numBooks;`
	`int numStudents = 40;`
	`inFS >> numBooks;`
	`cout << "Books per student: ";`
	`cout << numBooks / numStudents;`
		The file stream is not big enough.
		The file stream has not been properly opened.
		The >> operator cannot be used here.
		The code is fine.

2	Which statement opens a file outfile.txt given `ofstream outFS;`	`ofstream.open.("outfile.txt");`
		`outFS.(outfile.txt);`
		We need to declare an ifstream not an ofstream.
		`outFS.open("outfile.txt");`
		`outFS.("outfile.txt");`

3	Given the following code, which correctly writes "apples" to file outfile.txt? `ofstream outFS;`
	`outFS.open("outfile.txt");`
		`outfile.txt << "apples";`
		`outFS << "apples";`
		`outfile.txt >> "apples";`
		`outFS >> "apples";`
		`ofstream << "apples";`

4	Given the following code, which correctly opens the file?
	`string myfile = "outfile.txt";`
	`ofstream outFS;`
		`ofstream.open(myfile);`
		`outFS.open(myfile);`
		`outFS.open(myfile.c_str());`
		`outFS.open(c_str("outfile.txt"));`
		`outFS.open(outfile.txt);`

5	Given `ifstream inputFile;`, which statement correctly executes until the end of the file?	`ifstream.open(myfile);`
		`while (inputFile.eof()){`
		`while (!inputFile.eof()){`
		`while(inputFile.is_open()){`
		`while(inputFile.close()){`

Exploring further:

- fstream Reference Page from cplusplus.com

Section 9.6 - Stream errors

Sometimes user input, or file content, causes the stream to enter an error state. Ex: User enters `two` when an integer is expected. A ***stream error*** occurs when insertion or extraction fails, causing the stream to enter an error state.

An input stream may enter the error state because the value extracted is too large (or small) to fit in the given variable. While in an error state, an input stream may: skip extraction, set the given variable to 0, or set the given variable to the maximum (or minimum) value of that variable's data type.

Figure 9.6.1: User inputs string instead of integer, causing stream to enter an error state.

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

int main() {
   int num1 = -1; // Initial value -1 for demo purposes.
   int num2 = -1;

   cout << "Enter a number: " << endl;
   cin  >> num1; // Stream error state entered here.

   cout << "Enter a second number:" << endl;
   cin  >> num2; // Stream already in error state, so extraction skipped.

   cout << "num1: " << num1 << endl;
   cout << "num2: " << num2 << endl;

   return 0;
}
```

```
Enter a number:
six
Enter a second number:
num1: 0
num2: -1
```

A stream's error state can be checked with a function. Ex: `cin.good()` returns true if cin is not in an error state. Otherwise, false is returned.

Table 9.6.1: Functions to check error state.

Function	Meaning
good()	No error.
eof()	End-of-file reached on extraction.
fail()	Logical error on extraction or insertion.
bad()	Read (or write) error on extraction (or insertion).

A stream's error state is cleared using clear(). Ex: `cin.clear()` clears the error state from cin.

The function ignore(maxToIgnore, stopChar) ignores characters in the stream buffer. Ex: `cin.ignore(10, '\n')` ignores up to 10 characters in the stream buffer, or until a '\n' character is encountered.

Commonly, a program needs to wait until a '\n' character is found, in which case set maxToIgnore to the maximum size of a stream: `numeric_limits<streamsize>::max()`.

Figure 9.6.2: Read user input until a number is entered.

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

int main() {
   int number = 0;

   cout << "Enter a number: " << endl;
   cin >> number;

   while (cin.fail()) {
      // Clear error state
      cin.clear();

      // Ignore characters in stream until newline
      cin.ignore(numeric_limits<streamsize>::max(), '\n');

      cout << "Try again: " << endl;
      cin  >> number;
   }

   cout << "You entered: " << number << endl;

   return 0;
}
```

```
Enter a number:
six
Try again:
6
You entered: 6
```

A program may need to check for errors during file reading. One approach is to check whether end-of-file was reached after the file reading ends. If

end-of-file was not reached, then an error in file reading occurred.

Figure 9.6.3: Check for errors while reading a file.

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

int main() {
   ifstream inFS;
   int fileNumber = 0; // Number in file

   inFS.open("myfile.txt");

   if (!inFS.is_open()) {
      cout << "Could not open file myfile.txt." << endl;

      return 1;
   }

   // Read file until end-of-file or an error
   while (inFS.good()) {
       inFS >> fileNumber;
       cout << "File number: " << fileNumber << endl;
   }

   // If end-of-file not reached, then an error occurred
   if (!inFS.eof()) {
      cout << "Error reading myfile.txt" << endl;

      return 1;
   }

   inFS.close();

   return 0;
}
```

myfile.txt:

```
5
8
six
4
6
```

```
File number: 5
File number: 8
File number: 0
Error reading myfile.txt
```

Participation Activity 9.6.1: Stream error functions.

#	Question	Your answer
1	Which returns whether a logical error occurred with cin?	cin(fail)
		cin.fail()
		fail(cin)
2	Which function wouldn't be used to check cout's error state?	good()
		bad()
		eof()

Section 9.7 - Extraction before getline

The getline() function and the extraction operator >> handle a trailing newline differently, which can lead to a problem.

- The getline() function reads a line of text from a buffer, discarding the ending newline character.
- The extraction operator >> skips whitespace, then reads the next item such as an integer or string which is said to end at the next whitespace, leaving that ending whitespace character in the buffer (an exception being for reading a single character).

The problem is that code like `cin >> myInt;` and `getline(cin, nextLine);` may not behave as expected if the integer is ended with a newline. The getline() function will read that single remaining newline character, returning an empty string, rather than proceeding to the next line.

A simple solution is to not mix the two approaches to reading an input buffer, either only using extraction, or only using getline().

If one must mix the two approaches, then after an extraction operation, the trailing newline should be discarded from the buffer before calling the getline(), by inserting some statement in between. One possible solution inserts cin.ignore(), which discards the next character in the input buffer. Another possible approach inserts another getline() call, ignoring its blank string.

Participation Activity 9.7.1: Using extraction and getline.

Given the following user input, what is the content of the string resultStr for each of the following code segments. Include quotes "" in your answer. If resultStr is an empty string, answer "".

```
Bob Thomas, AZ, 31
Fred Robahauck, NY, 45
```

#	Question	Your answer

1	`string firstName;`
	`string lastName;`
	`string city;`
	`int playerNum;`
	`string resultStr;`
	
	`cin >> firstName;`
	`cin >> lastName;`
	`cin >> city;`
	`cin >> playerNum;`
	`getline(cin, resultStr);`	

2	`string firstName;`
	`string lastName;`
	`string city;`
	`int playerNum;`
	`string resultStr;`
	
	`cin >> firstName;`
	`cin >> lastName;`
	`cin >> city;`
	`cin >> playerNum;`
	`cin >> resultStr;`	

3	`string firstName;`
	`string lastName;`
	`string city;`
	`int playerNum;`
	`string resultStr;`
	
	`cin >> firstName;`
	`cin >> lastName;`
	`cin >> city;`
	`cin >> playerNum;`
	`cin.ignore();`
	`getline(cin, resultStr);`	

Exploring further:

- getline Reference Page from cplusplus.com