Basic C++ Stream I/O

David Kieras, EECS Dept., Univ. of Michigan
12/29/2011

Stream Basics

What's a stream?

A stream is a popular concept for how to do input/output. Basically, a stream is a sequence of characters with functions to take characters out of one end, and put characters into the other end. In the case of input/output streams, one end of the stream is connected to a physical I/O device such as a keyboard or display. If it is a console output stream, your program puts characters into one end of the stream, and the display system takes characters out of the other and puts them on the screen. If it is a console input stream, the keyboard puts characters into one end of the stream, and your program takes characters out of the other and stores the results in variables in the program. If no characters are waiting in the input stream, your program must wait until you supply some by typing on the keyboard. File streams follow the same principle, except that the file system is attached to the other end of the stream.

What are `cin` and `cout`?

Two streams exist to allow you to communicate with your "console" - the screen and keyboard setup that makes up the traditional computer user interface. These are the input stream `cin` (console input), which is connected to the keyboard, and the output stream `cout` (console output), which is connected to your display. These two streams are created when the system starts your program. To use `cin` and `cout` and the input and output operators for them, you have to tell the compiler about them by including the relevant declarations in the library header file:

```
#include <iostream>
using namespace std; // See the namespace handout for when this form is appropriate
```

When your program is linked, the relevant library object modules must be included - usually this is automatic or a normal part of your compilation/linking process. The streams `cin` and `cout` are actually global variables (or global objects) that are declared in `<iostream>` and initialized during program startup.

This document uses `cin` and `cout` for examples, but everything herein works for any other text input or output stream, such as one your program might create itself for disk file input/output.

Stream Output

How does the output operator work?

The behavior of the output operator << resembles the C `stdio`'s `printf` function. The value of the variable is converted into a sequence of characters depending on the type of the variable and a set of formatting rules, and these characters are inserted into the output stream for transmission to the output device (thus << is also called the stream *insertion* operator). Additional items, called *manipulators*, can be output to control the output stream. For example:

```
int int_var = 123;
cout << int_var << endl;
```

The variable `int_var` is an integer containing the binary value for 123. The output operator interprets this bit pattern for an integer variable as meaning it should construct the sequence of characters '1' , '2', and '3' and insert them into the output stream. When the next output operator gets the *manipulator* `endl` , it adds a newline character (`'\n'`) and flushes the output stream - forces it to write all of its characters to the display. The manipulator `flush` will flush the output stream without writing a newline character. The reason why flushing is involved is that the output stream is buffered for greater speed - instead of sending each character to the output device individually (which would be very slow), the system normally stores many characters in memory and when appropriate, writes them all out at once.

In the following two examples, the bit patterns in the variables are interpreted to cause the character 'Z' to appear on the screen for the character variable, and the sequence of characters '-', '3', '.', '1', and '4' in the case of the double-precision floating point variable (this type is called "double" in C and C++; single-precision floating point is called "float" and is rarely used because its precision is too low for any serious numerical work).

```
char char_var = 'Z';
cout << char_var << endl;

double double_var = -3.14;
cout << double_var << endl;
```

What if you have a constant instead of a variable? The compiler figures out the type of the constant from its appearance, and then the output operator acts accordingly. For example, the following will produce the same result as the previous three examples:

```
cout << 123 << endl;              // an integer constant
cout << 'Z' << endl;              // a character constant
cout << -3.14 << endl;     // a double constant
```

Outputting "C-strings" or arrays of characters is more complicated; it works like C's `printf` with `"%s"`. If the type of variable being output is of type `char *` (pointer to characters), then the output operator copies all of the characters starting at the pointed-to address into the stream up to, but not including, the null byte (`'\0'`) that by C convention is used to terminate a so-called "C string." A string literal constant is converted into a pointer to the C-string placed somewhere in memory by the compiler, where it can be used to initialize a `char` array, or just serve as a character string constant. So the first output statement below results in "string literal" being displayed, while the second results in "array of chars":

```
cout << "string literal" << endl; // output a char * for a string constant

char str[20] = "array of chars";
cout << str << endl;              // output a char * where an array starts
```

You can output more than one value in the same statement - the output operator *cascades* from left to right, putting the sequence of characters for each variable in the output stream. You can wait until all variables have been put into the stream before putting in the endl manipulator. For example, using the variables from the above example:

```
cout << "The integer and char are " << int_var <<  ',' << char_var; // no endl
cout <<   " and the double is " <<  double_var << '!' << endl;
```

This will produce the output:

```
The integer and char are 123,Z and the double is -3.14!
```

Notice how there is no space before or after the comma between "123" and "Z", or before the exclamation point at the end.

What format is the output in?

Unless you make use of the formatting controls for output streams, the stream output operator uses a standard default layout for different data types. The basic principle is to produce as many characters as necessary to adequately represent the output, but no more. Thus, for example, all of the digits of an integer will always be output, and so will all the characters in a string. Double values are more subtle because you usually don't want to see all of the places to the right of the decimal point. According to the C++ Standard rules, up to 6 digits will be produced, counting both to the left and right of the decimal point, but fewer if the result is accurate, or scientfic notation if the number is too big to represent with 6 digits after dropping all the places to the right of the decimal point. The results of these default formatting rules are usually just fine, unless you want a bunch of numbers to line up neatly; if so, you must use the formatting controls described in the Output Formatting document.

The error output stream

The output buffering mentioned above is why debugging with ordinary output statements can be very misleading about when an error occurs - the output often lags far behind the program execution; a program can crash before the last output gets written. If you want to use "printer debugging", try using the console error stream `cerr` instead of `cout` (The corresponding stream in C is named `stderr`). `cerr` is a stream that behaves like `cout` except that it is not buffered - every character sent to `cerr` is immediately output to the display. The result is very slow, but more suitable for debugging purposes where a program crash might follow. `cerr` is declared and initialized along with `cin` and `cout`, so you don't have to do anything additional to use it.

Stream Input

How does the input operator read in values? The behavior of the input operator >> resembles the typical use of the C `stdio` `scanf` function but it is more consistent and safer. Input is more complicated than output, because the situation is more complicated — for example, the user might type garbage, so the input functions have to deal with it.

The input operator works according to the following process:

1. If there are no characters waiting in the input stream, wait until the system says there are characters ready to be read. This is an important point: normally, the system lets the user type characters, and saves them without letting the program see them. This is to give the user a chance to backspace and correct any typing errors. When the user is finishing typing, he or she signals that the input is ready to be processed by hitting the Carriage Return (or Return) key. The system then allows the input operator to start processing the input.

2. Then the input operator starts scanning the characters in the input stream. It always skips any initial whitespace characters. Whitespace characters are characters like space, carriage return, or tab - when typed or displayed, their effect is simply to make some "white space" between other characters. Then it tries to find a sequence of characters that satisfies the input request. It goes as far as it can in the input, and stops when the request is satisfied, or if it encounters a character that indicates the end of the desired input.

3. All characters used to satisfy the request are "consumed" and removed from the stream. The next input operation will start with the first character that was not used.

Thus a value is extracted one-character-at-a-time from the input stream (the >> operator is also called the stream *extraction* operator).

Example:

```
int int_variable;
cin >> int_variable;
```

The request is to get an *integer* from the input stream. The input operator scans the input stream from the first character present, and skips over any whitespace. If the first non-whitespace character can be part of an integer (a digit, '+', or '-'), the scan will collect that character and continue collecting characters until it encounters a character that cannot be part of an integer, such as a letter or whitespace. All of the characters collected will then be used to determine the value stored into int_variable. If the first non-whitespace character can not be part of an integer, the input operation fails, and nothing is put into the variable.
Suppose the user supplied the following input, where <CR> means the carriage return character, and the whitespace indicates one or more blank characters:

```
  123     <CR>
```

The leading whitespace is skipped, the characters '1', '2', and '3' are collected and the scan stops at the first whitespace after '3' because a space can't be part of an integer. The collected characters specify a value of 123 and this is stored in int_variable. The request is satisfied, and the next input operation will start at the space after the '3'.

You can input several variables in a single statement by *cascading* the input operator. For example, suppose the program contained:

```
cin >> int_variable1 >> int_variable2 >> int_variable3;
```

and the input from the user was

```
  123 456 789
```

The value 123 would be read into `int_variable1` by the first input operator, 456 into `int_variable2` by the second, and 789 into `int_variable3`, by the third input operator; the operators are executed left-to-right.

Since whitespace is skipped over, you are free to break up the input into multiple lines, or put multiple items on the same line, as long as you have at least one whitespace character to separate them. Thus if the above program statement, or even three separate input statements, processed the following line:

```
    123 456           789 <CR>
```

exactly the same result would be obtained by reading from the following two lines

```
123 <CR>
              456           789 <CR>
```

or the following three lines:

```
123                 <CR>
456 <CR>
                     789<CR>
```

Now, if the user types something that does not satisfy the input request, an error will occur when the input operator tries to scan the input. The tricky problem is that it is possible for the user to make typing errors whose results happen to satisfy this input request, but mess up the next input operation. Input errors will be discussed later in this document.

How does the input operator work for different variable types?

For the most common built-in types, `cin >>` behaves as follows, where the variable names indicate how they are declared:

The char type:

```
cin >> char_variable;
```

Skip whitespace, read the first non-whitespace character into the char variable. The character that follows will be the first one read next time. Because whitespace is skipped first, the behavior is **not** identical to scanf's "%c" format. In fact, there is no way to use the input operator >> to unconditionally read the next character regardless of whether it is whitespace (the `get` function is used for this; see below).

The int type:

```
cin >> int_variable;
```

Skip whitespace, read all characters that could make up an integer, stop scanning when the next character cannot be part of an integer (e.g. whitespace, a letter, a period). The next input scan will start with that character. This is like scanf's "%d".

The double type:

```
cin >> double_variable;
```

This is the same as reading into an int, except that characters such as a decimal point can be part of the double value. This is like scanf's "%lf".

Note on inputting double or floating-point variables: If you read input into a double variable, the user does not have to enter a decimal point. That is, "123" can be a value for a floating-point variable just as much as "12.3" can. As far as input processing in C and C++ is concerned, "123" can be either read into a double, or read into an integer. This fact causes some confusion to beginning programmers — the typed-in data values can all be integer-like in appearance, and still be read into double floating-point variables. This ancient practice, dating from the earliest days of FORTRAN, saves needless and error-prone typing of decimal points if the value happens to be a whole number.

The char * type:

```
cin >> char_pointer_variable;
```

where the variable is of type `char *`

This reads a C-style string, like `scanf's "%s"`. Whitespace is skipped, then input characters are copied into the array of characters that `char_pointer_variable` is assumed to point to. Copying stops when another whitespace is encountered, and this character will be the first one read in the next input operation. A `'\0'` character is placed in

the array after the last character from the input in order to form a valid C string. As usual with C/C++ arrays, no check is made for whether the array is big enough, so this input form is dangerous. Be sure to declare the array big enough to hold anything that the user might possibly type, and pray that they don't type more! An example of typical usage (which is still dangerous):

```
char input_buffer[32]; // array has space for 31 characters - the null byte takes one!
...
cout << "Enter a string no more than 31 characters long!"  << endl;
cin >> input_buffer;       // no protection against overflowing the array
cout << "You entered: " << input_buffer << endl;
```

Fortunately, there is a somewhat clumsy way to limit how many characters get read and stored, analogous to how `scanf`'s `%s` formatting item works. You can use a member function of an input stream to set the maximum width for the next read into a C-string, or an equivalent manipulator. Here is an example:

```
#include <iostream>
#include <iomanip>

using namespace std;

{
    char str[10];
//   using input stream member function:
//   cin.width(10);   // store a maximum of 10 characters
//   cin >> str;

//   using stream manipulator:
     cin >> setw(10) >> str;  // store a maximum of 10 characters
}
```

Once the stream width has been set, the input operator will read and store up to that many characters minus one (9 in this example), and then store a null byte in the last character. If you want to read another C-string, you must set the stream width again (it is not "sticky"). This way, you never overflow the array and always get a validly terminated C-string stored in the array. Any extra characters are left in the stream for the next read.

Note: You must allocate space for the string, either in the form of an array of characters, or a piece of dynamically allocated memory. A common error is to read with a char * pointer that does not point to any such allocated space; as in the following:

```
char * buffer; // uninitialized pointer - always dangerous!
cin >> buffer; // UNDEFINED BEHAVIOR - buffer does not point to reserved space!
```

The result is *undefined* - no promises are made about what will happen when characters are stored at the garbage address that is probably in the pointer variable. You may overwrite something critical in memory, crashing the program or system, or you might get "lucky" and scribble on some space that is unused at the time, but crashes the program later, or in a different run, or on a different machine or system, or in a different century or alternate universe. Beware!

There are better, safer ways to input strings, namely using std::string, which automatically expands its internal space as needed to hold the input, and throws an exception if memory is exhausted.

What if the input isn't correct?

If the input characters are not consistent with the input request, the request will not be satisfied. This can be due to errors in the input data (e.g. typing errors), or incorrect assumptions in how the program is written - it might be demanding an integer when the user and programmer quite reasonably believe that a double is appropriate. This situation is called **invalid input** in this document. Because humans make lots of typing errors, invalid input is very common.

This document describes how to detect and deal with simple input errors from keyboard input; file I/O involves additional issues. Usually, you will print out a message to the user that there has been an error in the input, do something to clean up the input stream, and then continue processing if possible.

What happens if the input is invalid?

The stream input operator results in a failure only if it is unable to find *anything at all* that satisfies the input request. Depending on exactly what is in the input, the input operator might succeed, but **not** produce the intended result. For example, suppose the program executes this statement:

```
cin >> int_variable;
```

and the user made the following various typing errors:

```
    a123    <CR>
```

This will fail because an integer cannot start with an 'a'. The value of int_variable will not be altered in any way. The scan stopped at the 'a', so the next input will start there. As a result, simply re-executing the same input statement will fail again because the same "a" will be scanned first. This is why you usually have to clean up the input stream before continuing.

```
    12a3  <CR>
```

This will succeed but the value stored in int_variable will be 12, which may not be what the user intended. The scan consumed the '1' and the '2' and stops at the 'a' because it can't be part of an integer, and the next input will start there. As far as C++ is concerned, this is a perfectly normal situation; as far as it knows, maybe the next character is supposed to be an 'a.' If a three-digit integer was expected, your code must check the value and decide what to do. If the next input is to read another integer, then there will be a failure on the 'a'. So the first input operation can succeed, but produce a wrong result, and the problem might not show up until a later input. It is up to your code to check for and cope with the situation.

```
   12.3  <CR>
```

The result is the same as above; int_variable gets set to 12. The next input scan will start at the '.' character. If the user thinks that a non-integer is a valid input here, and your program tries to read an integer, both of you are going to be unhappy. *Note:* The above examples use reading numbers, because there are no invalid characters if you are reading a character or a string!

How do you tell whether the input operation succeeded?

The system maintains information about what state a stream is in. If the input operation succeeds, the stream will be in a **good** state; it it failed, the stream will be in a **non-good** state (file streams can fail in several ways). The stream itself, e.g. `cin`, can be tested directly - if the stream is still in a **good** state, the stream will test as true (non-zero); if the stream is not in a **good** state, it will test as false (zero). So you can easily check whether there is a problem, but you have to do more to determine what the nature of the problem is.

Here is the simplest pattern for checking whether input succeeded:

```
cin >> input_variable;
if (cin) {/* stream is good, input was valid and can be used */}
or
if (!cin) {/* stream is not good, nothing placed in input_variable */}
```

This can be folded into one statement, since value returned by the input operator is the stream itself:

```
if (cin >> input_variable) {/* stream is good, input was valid */}
```

In fact, the whole input expression is often used to control the reading loop. This is a common idiom in C++:

```
while (cin >> var1 >> var2)
{/* stream is good, input was valid */}
```

What do you do if the input was invalid?

It is important to remember that once the stream is no longer **good**, is in a failed state and it will stay that way, and any additional input operations will do **nothing**, no matter what they are or what is in the input. You have to **clear** the stream state before input will work on the stream again. If you don't clear the stream state, your program will drop through all the remaining input statements doing nothing; often, your program will appear to hang, or loop forever. While this may seem cranky, it is a way to ensure that if your program gets incorrect input, and your code does not detect and handle it appropriately, something obviously wrong will probably happen. The simplest solution to invalid input is to simply let the user "kill" or cancel the program and start over, but needless to say, this is annoying for the user!

OK, how to do I handle an input error?

First, you can clear the stream state with the member function `clear()`:

```
cin.clear(); // reset the input stream state
```

The name is misleading - "reset" would have been better. The `clear` function simply resets the stream state by resetting the error flag bits in the stream object; it does *not* change what is waiting in the stream of characters, and so is usually not enough to recover from an input error. The input scanning operation stopped where the incorrect character was encountered. If you try to read the stream again, you will be trying to read that same character, and so even if you clear the stream state, the same problem will happen if you try to read the same thing. Usually, you want to print a message for the user, clean out the bad input, and then start fresh. So a typical pattern would be:

Attempt to read some input.
Check the stream state.
If the state is **good**, process the input.
If the state is **not good**, do the following:
    Print a message informing the user of the bad input.
    Clear the stream state with the `clear()` function.
    Skip the offending material.
    Resume processing.

How do you skip the offending material? It depends on what it is and what your program needs to do. A good general-purpose approach used in this course is to skip the rest of the input line by simply reading all of the characters up to and including the newline character (`'\n'`) at the end of the line. However, there is a complication: You can't just do a bunch of reads into an character variable, because the newline will get skipped automatically (it's whitespace!) so you will never know when it has been encountered! Instead, use the `get` member function, defined below, and used in the complete example at the end of this document.

```
int get();    // prototype
char_variable = cin.get(); // example
```
Reads the next character, even if it is whitespaee, and returns it (remember int's convert to char's).

A handy way to read all the characters up to and including the next newline takes advantage of C/C++'s reputation for terse code:

```
while (cin.get() != '\n');
```

This one line of code first calls the cin.get() function to get the next character from the stream. The returned value is compared to the newline character. If it is not the same, the condition of the while loop is true, so the empty body of the while loop is executed, and the condition of the while loop is tested again with a new call to cin.get(). When the newline is read, the condition becomes false, and the loop terminates. The next character in the stream is now whatever followed the newline character.

There is also a member function called `ignore` some people use for this purpose, but it requires specifying the maximum number of characters to be skipped, which is ugly or could even be misleading in this case - how do you know how long the rest of the line would be? In fact, the only good way to use `ignore` to skip the rest of the line, regardless of how long that line might be, is to specify the maximum number of characters that the implementation allows to be in the stream. This value is supplied in the `<limits>` header, as a specialization of the `numeric_limits` template with `streamsize`. Using it in a call to ignore would look like this:

```
#include <iostream>
#include <limits>
using namespace std;
// other code
// skip over the input until a newline is encountered or end-of-file is reached
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

For purposes of this course, you should either use the `while` loop with `cin.get()`, or the fully-specified `cin.ignore` call shown above. Anything else is likely to be awkward, misleading, incorrect, or egregiously inefficient.

What about errors in cascaded input operators?

You can input several variables in a single statement by cascading the input operator. Since the stream state remains until cleared, you can check the state of the stream at any time after any number of attempted input operations. For example, suppose the program statements were:

```
cin >> int_variable1 >> int_variable2 >> int_variable3;
if (cin) {/* continue processing */}
else {/* invalid input */}
```

and the input from the user was

```
  12 a34  56
```

Here's what would happen. The first application of operator>> will successfully read the "12" and store the value in `int_variable1`. The second application of operator>> will fail because the 'a' is the first non-whitespace that comes next, and it can't be part of an integer; nothing will be put into `int_variable2`. This failure means the stream is no longer good. The third application of operator>> will do **nothing**, because the stream is not good. Nothing will be put into `int_variable3` either. The check of the stream state will then tell you that the stream is no longer good, but you can't tell by this check which of the three input operations failed. So if the stream is not good after a cascaded input statement, you should assume that none of the variables have valid values.

Should I check for `EOF` in console input?

Some students think they should be testing `cin` for end-of file (`EOF`) after every input, as in:

```
cin >> int_var;
if(cin.eof())
{/* time for the program to quit, or something is wrong */}
```

This is **not** a customary or normal way to handle the console input stream; it is not *idiomatic*. Only in very unusual situations is keyboard input terminated with an end-of-file condition. One reason is that the keystroke that creates an end-of-file condition from the keyboard is not standard, but differs between platforms. Another reason is that terminating the program is best done more explicitly (e.g. with a "quit" command). A third reason is that controlling a reading loop with `eof()` is always incorrect, even if a file stream is involved (see the File Stream handout) - because other error conditions might arise, and `eof()` will not detect them. In this course, testing `cin` for `eof()` will be considered a mark of poor coding practice, and penalized, so don't do it.

Example of Detection and Handling of Invalid Input

The following example program calls a function named `get_int` which asks the user for an integer, and insists on it, not returning until it gets one. If the user types in a valid integer, `get_int` skips the rest of the line and returns the value. If the user fails to input something that can be treated as an integer, `get_int` displays a message, skips the rest of the line, and requests input again.

The input operator expression as a whole is tested in the `if` statement, to see if the stream tests as zero or not. If it tests as zero, then something went wrong in trying to find an integer. The `clear()` function is called to reset the stream state. Then the `get()` function is used to read past any characters left on the line. If the input operator expression tests as non-zero, an integer was successfully read, and the rest of the line is skipped.

Examine the sample output. Notice how garbage after a string of digits is OK - it just terminates the integer, then gets skipped. But garbage in front of the integer prevents any value from being found. In this case, the `get_int` function prints an error message, skips the entire remainder of the line, and waits for another input.

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

int get_int();             // prototype

int main ()
{
       cout << "Enter five integers" << endl;
       for (int i = 0; i < 5; i++)
```

```
              cout << i+1 << ". You entered " << get_int() << endl;
       cout << "Thank you for your cooperation!" << endl;
       return 0;
}

// ask the user for an integer, and return it; insist that the user provide one
int get_int()
{
       int x;
       bool done = false;
       do {
              cout << "\nPlease type an integer: ";
              if (cin >> x)       // Do the input, then check: is the stream good?
                     done = true;  // If so, input was good, and we are done
              else {
                     cout << "Could not find a valid integer." << endl;
                     cin.clear();  // clear the error bits
                     }
              cout << "Skipping rest of input line." << endl;
              // get next character until it is a newline, then stop
              while(cin.get() != '\n');
              }
       while (!done);      // repeat until we get a valid input
       return x;
}
 SAMPLE OUTPUT
Enter five integers

Please type an integer: 134
Skipping rest of input line.
1. You entered 134

Please type an integer: 56xyz
Skipping rest of input line.
2. You entered 56

Please type an integer: 23
Skipping rest of input line.
3. You entered 23

Please type an integer: x13yz
Could not find a valid integer.
Skipping rest of input line.

Please type an integer: 13
Skipping rest of input line.
4. You entered 13

Please type an integer:

   55a55
Skipping rest of input line.
5. You entered 55
Thank you for your cooperation!
```