A Bit More About I/O in C++

Member Functions of the `istream` Class

The extraction operator is of limited use because it always uses whitespace to delimit its reads of the input stream. It cannot be used to read those whitespace characters, for example. The `istream` class, has an assortment of other methods of input that are useful for different types of tasks. Remember that all input file streams (`ifstream` objects), all `iostream` objects (`cin`, `cout`, and `cerr`), and all input string streams (`istringstream` objects) are types of `istream` objects, meaning that they "inherit" all public member functions (and other public and protected members) from `istream`. This is a very short tutorial on some of the member functions of the `istream` class that may prove to be useful.

The get() Member Functions

The `get()` member function has six overloaded versions, but only four will be discussed here. The first two versions are for reading single characters at a time.

One Character at a Time

The first of these is

```
int get();
```

This extracts a single character from the stream and returns its value cast to an integer. It does not return a `char` because the `char` type is unsigned, and when get fails, it might return -1, depending upon the implementation. `get()` reads every single character: blanks, tabs, newlines, and so on, one at a time. When there are no characters, the `eof` bit is set. This means that you can use this form of `get()` as follows.

```
int main()
{
        char c;
        while ( (c = cin.get()) && !cin.eof()  )
                cout << c;
        return 0;
}
```

This little program will copy the standard input stream to the standard output stream. If you compile this into a command named `copy`, for example, and type

```
copy < oldfile > newfile
```

it will make a copy of `oldfile` in `newfile`.

A second overload has the signature

```
istream& get ( char& ch );
```

This also extracts a single character from the stream, but does not return it. Instead it stores it in its call-by-reference parameter `ch`. Like the first, if the end of file is reached, it sets the `eof` bit and does not alter the value of `ch`. The preceding program could be written using this version of `get()` as follows.

```
int main()
{
        char c;
        while ( cin.get(c) && !cin.eof() )
                cout << c;
        return 0;
}
```

Reading Strings

The two remaining overloads can be used to read data into ***C strings***, ***not C++ strings***! Unlike the versions that read one character at a time, *these reads are delimited.* This means that they read until either they read a certain number of characters or they see a delimiter character, which is by default a newline, and they do not remove that delimiter from the input stream. Because the delimiter is not removed from the input stream, the next time they are called on that stream, they will read nothing because the delimiter is the next character they would read. Therefore, for the next call to read beyond the delimiter, it must be removed by some other means. These functions are really more useful when it is expected that the delimiter *will not be found*, and that finding the delimiter is the exceptional case. The examples will clarify this point.

There are two prototypes:

```
    istream& get (char* str, streamsize n );
    istream& get (char* s, streamsize n, char delim );
```

The only difference between these is that the first uses the newline character as the delimiter and the second lets the caller choose the delimiter. They both extract characters from the input stream and store them into the C string beginning at `str`. Characters are extracted until either (`n - 1`) characters have been extracted or the delimiting character is found. In other words, if the delimiter is reached before `(n-1)` characters have been read, reading stops, or if `(n-1)` characters have been read but the delimiter has not been reached, reading also stops. In either case the string is padded with a `NULL` byte after the last character. The extraction also stops if the end of file is reached in the input sequence or if an error occurs during the input operation. If `get()` extracts no characters, *it sets the failbit on the stream.*

If the delimiter is found, it is not extracted from the input sequence. The next call to this version of the `get()` function will return nothing, will not remove it, and will set the failbit to true preventing any further input operations on the stream. If you want to force this character to be extracted, you must use the `getline()` member function.

The point of providing a second argument is to prevent the C string from being overflowed. You should always set it to no more than the size of the array pointed to by the first argument.

The number of characters read by any of the previous input operations can be obtained by calling the member function `gcount()`. The `gcount()` function is useful when using the string-reading versions of `get()` because you can test when they have stopped reading anything. The next program will read a single line of text, breaking it up into chunks of size 10 with a `':'` between the chunks.

```
int main()
{
    char str[10];
    cin.get(str, 10);
        while ( cin.gcount() > 0 ) {
        cout << str << ":";
        cin.get(str, 10);
    }
    cout << endl;
        return 0;
}
```

The following program shows how to use `get()` properly when either stopping criterion may occur: a delimiter was found or (n-1) characters were read. It uses `getline()` to read the delimiter, and it clears the failbit when it reads nothing.

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

#define CHUNKSIZE  14

int main()
{
    char str[CHUNKSIZE];
    char junk[2];
    istringstream iss;
    int i = 0;

    string strdata =
    "12345678901234\n"
    "123456789012345678\n"
    "123456\n"
    "1234567890123\n"
    "123456789012\n"
    "1234567890123456789012345678\n"
    "12345678901\n"
    "123456789012345678901\n"
    "12345678901234567\n"
    "1234567890123\n";

    iss.str(strdata);  // Make iss a stream like a text file with this string

    while ( !iss.eof() ) {
        iss.get(str, CHUNKSIZE);
        // if get() read no characters, it set the failbit, so we clear it
        // Otherwise we print what we got
        if ( iss.gcount() == 0 )
            iss.clear();
        else
            cout << i << "  " << str << endl;

        // If gcount < CHUNKSIZE-1 we found the delimiter, so remove it
        if ( iss.gcount() < CHUNKSIZE-1 )
            iss.getline(junk, 2);
        i++;
    }
    cout << endl;
    return 0;
}
```

The output is the sequence of chunks that are read by successive calls to `get()`. A line of output can never be larger than the chunk size minus one, and will be smaller when the newline is found first. After each call to `get()` we check whether `get()` read nothing. If it did, it set the failbit, and no input will be possible on the stream after that, so we must clear it.

After that we check whether (`CHUNKSIZE-1`) chars were read. If less, it found the delimiter, so `getline()` is used to extract it. Because the last character in the stream is a newline, `getline()` will read it. The next call to `get()` will read nothing, and `iss.gcount()` will be less than (`CHUNKSIZE-1`), causing `getline()` to run again. This time `getline()` will encounter the end-of-file condition and set the eofbit on the stream and the loop will exit.

The getline() Member Functions

The `getline()` functions that are members of the `istream` class are not the same as the global `getline()` function. The ones that are part of `istream` are called on an `istream` object, just like the `get()` functions above, e.g.

```
char str[10];
cin.getline(str, 10);
```

The global `getline()` function is called with a stream object as its argument, and has a C++ string argument:

```
string s;
getline(cin, s, 10);
```

It is easy to get these confused. The distinction, once again:

- ***member function needs C string***
- ***global function needs C++ string***

The `getline()` member functions of the `istream` class are similar to the `get()` member functions that take a C string argument, except that they extract the delimiter and they set the failbit under different conditions. Their prototypes are

```
istream& getline (char* str, streamsize n );
istream& getline (char* str, streamsize n, char delim );
```

Unlike the `get()` functions, these work nicely for reading text files line by line, and they are suitable for any type of reading tasks where the text is delimited and the delimiter must be read and discarded. The only caveat is that the size of the read should be large enough that the delimiter is found within the first (n-1) characters, otherwise you have to deal with the failbit being set. Thus, if you know the line size, for example, is never larger than 200 characters, call it with

```
cin.getline(str, 200);
```

Now the details. Calling

```
input.getline(str, n, delim) ;
```

where `input` is an input stream, `str` is a C string, and `delim` is some character used as a delimiter, will extract characters from the stream until one of three conditions occurs:

1. end-of-file occurs on `input`, in which case it sets the eofbit on the `input` stream;
2. the delimiter `delim` is the next available input character, in which case the delimiter is extracted but not stored into `str`;
3. `n` is less than one or `(n-1)` characters are stored into str, in which case the failbit is set on the `input` stream.

These checks are made in the stated order. This way, if there are `(n-1)` characters followed by a delimiter, it will not set the failbit, because it first finds the delimiter before checking whether `(n-1)` characters have been stored.

The implication of the above is that when you use `getline()`, you have to check whether it stopped reading because it found the delimiter, or it stopped because it read `(n-1)` characters, or it reached end-of-file. The following examples show how to do this.

The first program reads from an input file specified on the command line and outputs a copy of that file on the standard output, i.e., the terminal.

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

/* copy from file argument to stdout*/
int main(int argc, char* argv[] )
{
    char buffer[10];
    ifstream fin;

    if ( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " file\n";
        exit(1);
    }

    fin.open( argv[1]);
    if ( !fin ) {
        cerr << "Could not open" << argv[1] << " for reading.\n";
        exit(1);
    }

    fin.getline(buffer, 10);
    while (  !fin.eof()  ) {
        cout << buffer;
        if ( fin.fail() )
            fin.clear();
        else
            cout << endl;
      fin.getline(buffer, 10);
    }
    cout << endl;
    return 0;
}
```

It uses a string of size 10, which we will call `buffer`, to store the characters read by `getline()`. Therefore, we repeatedly call `getline()` with 10 as its `streamsize` argument.

After the appropriate error-checking, it enters a loop in which it checks for end-of-file upon entry, and repeatedly calls `getline()`. It checks whether the failbit was set by the immediately preceding call. If it was not set, this means that `getline()` found the delimiter, so the program prints a newline (since `getline()` does not store it into `buffer`). If the failbit was set, it implies that 9 characters were read but no delimiter was found, and all we have to do is unset the failbit using the `clear()` member function of the `istream` class. It cannot be the case that we reached end-of-file, otherwise we would not be in the loop, so clearing the flags is safe here.

An alternative to the loop in the above listing is the following:

```
    while (  !fin.eof()  ) {
        fin.getline(buffer, 10);
        cout << buffer;
        if ( !fin.eof() && fin.fail() )
            fin.clear();
        else
            cout << endl;
    }
    return 0;
```

Instead of calling `getline()` before entering the loop and again at the bottom, we can call it upon entering the loop. But in this case we cannot simply clear the flags if the failbit is set, because the eofbit might be set by the last call to `getline()`. Therefore the condition for clearing is slightly different. Also, we can remove the last output of a newline when the loop ends, since that is now superfluous.