C++ Stream Input

Introduction

This handout provides a detailed description of the extraction operator, ">>", and also introduces some useful *member functions* of the *istream* class. These functions are made available when you include them by using "`#include <iostream.h>`", and if using Visual C++ .NET then the **.h** is left off and then add line of "`using namespace std;`", this is placed at the beginning of the program. Some header files in .NET do not need the **.h** after the file name.

More on the Extraction Operator

Recall that the ">>" operator is also sometimes called the *get from* operator or the *input* operator. It is actually part of the console input object, *cin*, which is declared in the header file, *iostream*. Console input using *cin* and ">>" can be done with *int*, *float* or any other standard C++ data type. In addition, C++ code can be written to overload ">>" to do input of programmer defined data types.

The rules that the extraction operator uses to process data from the input stream vary somewhat from type to type, but the rules for processing characters and numeric data are given below.

Numeric data:

1) Input is buffered, i.e. no characters are sent to the extraction operator for processing until a newline character is entered. The user is then allowed to make corrections by backspacing.

2) The extraction operator skips over leading whitespace characters, which are typically blanks, tabs, newlines, and form feeds, looking for appropriate numeric characters. If the first non-whitespace character is illegal, the extraction operator sets a flag and immediately returns to the function from which it was called. Program will continue to run, but no more input will be done until this error is correctd.

3) If first character is numeric character, then numeric characters are read and converted to the appropriate numeric type. Processing stops when the first non-numeric character is detected (often the newline). This character and any remaining characters are left in the input stream.

Character data:

1) Leading whitespaces are skipped.

2) Characters are processed until a whitespace character is encountered, which is left in the input stream.

The extraction operator keeps track of the current input state via a series of *flags* called "*iosflags*". Programmers can access these *flags* and in particular, can determine whether the last input operation was successful. This is done by checking the "*good*" flag via the function *cin.good()*, described later in this handout.

Avoiding Infinite Loops When Using the Extraction Operator

It’s relatively easy for a programmer to get into trouble when using the extraction operator. For example, consider the following loop.

```
do
{
  cout << “Enter a positive int =>”;
  cin >> N;
  if ( N <= 0 )
    cout << “You must enter a positive integer!\n”;
}
while ( N <= 0 );
```

Now suppose the loop is executed and the user types the characters below (“□” represents a blank, “↵” represents a newline character).

□□□-123x↵

The series of events goes something like this:

1) The extraction operator skips the leading blanks, then processes the ‘-‘, ‘1’, ‘2’, and ‘3’ characters, converting them to the int -123 and storing this int in N.

2) The ‘x’ character causes the “>>” to stop processing characters, because it is non-numeric. The ‘x’ does not cause a problem yet, but it does remain in the input stream.

3) The “`if ( N <= 0 )`” statement is executed and the error message displayed.

4) Control returns to the line “`cin >> N;`”, but the ‘x’ is still waiting to be read. This time the attempt of “>>” to read an int results in an error and the “good bit” is set to 0 (false). Control returns immediately from the extraction operator and the value of N is not changed, and the error message is again displayed.

5) Steps 3) and 4) are repeated and we have an infinite loop on our hands!

There are several ways to fix the loop problem above. See attached program for one method that uses functions *cin.good()* and *cin.clear()*. These functions and several other useful input functions are summarized on the pages that follow.

It is important to note that once an input operation fails, no more input will be processed until the input stream has been “reset”. In this class, we will usually take a very simple approach: We will write code that *assumes error free numeric input*.

Some useful istream Member Functions

In the examples below, where the prototype only is given, these member functions are called with an *istream object* followed by *dot operator* then *member name*, example is "`cin.get()`". Also assume that the following declarations have been made.

```
char Ch;
char Str[100];
```

`int` **`eof`**`()`

Returns a 1 if the *end of file* character has been encountered. (When reading from standard input in an MS-DOS environment, this means that the user has entered a Crtl-Z.)

`int` **`fail`**`()`

Returns a 0 if the last stream operation was successful.

`int` **`get`**`()`

A typical call to this function would be "`Ch = cin.get()`;". This call reads a character from standard input and is similar to using "`cin >> Ch`;". However, ">>" cannot read whitespace characters, since this operator always skips over these characters.

`istream&` **`get`**`(char &)`

This is an example of function overloading, since the function above is also named "`get`". The compiler, as well as human readers, can tell from context which function is intended. The call "`cin.get(Ch);`" is very similar to "`Ch = cin.get()`;". One difference is that this function returns the state of *istream*, so one can write loops such as

```
while ( cin.get(Ch) )   //while not eof
```

However, the loop can also be written with the other form using the condition

```
while ( (Ch = cin.get() ) != EOF )
```

`int` **`good`**`()`

Returns a true if the last istream input operation was successful. If `cin.good()` becomes false, your program should empty the input buffer and make a call to `cin.clear()` or no further stream input will be possible.

istream& **ignore**(int MaxChars = 1, char Terminator = EOF)

Discards characters waiting in the input stream. Stops when Terminator is encountered or MaxChars characters have been discarded. The call “`cin.ignore(100, ‘\n’);`” will read and discard characters until a newline is read or 100 characters have been read. Since buffered input is used by istream, no characters will be seen by *ignore* until the user presses the enter key. The call “`cin.ignore();`” used default parameters and will discard only a single character.

istream& **getline** (char *Str, int MaxChars)

Reads characters into the array Str until a newline is detected or MaxChars – 1 characters have been read. If the newline char is processed, it is counted as part of the MaxChars -1 total. In that case, the newline character is discarded and replaced with a null terminator. If there are more than MaxChars -1 characters preceding the newline character, then only the first MaxChars -1 characters are stored in Str (along with a null terminator). As an example, consider the call “cin.getline(Str, 5);”, which reads characters into the array Str. If “abc↵” is input, then “abc” is stored in the array Str. However, if “abcdef↵” is input by the user, then “abcd”is stored in the array Str (with the last location saved for the null terminator) and “ef↵” is left in the input stream. An optional third parameter can be used to specify an end of input character other then the newline character, i.e. “cin.getline(Str, 5, ‘t’);”, where input will stop at the first ‘t’.

int **peek**()

Returns the next character in the input stream, but does not remove it.

istream& **putback**(char Ch)

This can be used if one too many characters have been read from the input stream and one of them needs to be put back. This function is rarely needed.

int **gcount**()

Returns the number of characters read by the most recent get, getline, or read call. The most common use for gcount is to return the number of chars stored by getline(), which in most situations is cin.gcount() -1, because the newline character is not stored, but is replaced by the null terminator, which is not counted as a character read. However, be careful when using gcount with getline in situations where the max number of characters expected is reached or exceeded.

```
/*********************  cingood.cpp  ***********************************
Demonstrates some of the error handling functions in istream, showing how
they can be used to do a bombproof integer read.

The program displays a prompt, then attempts to read an int from standard
input.  If an error occurs, a message is displayed and the user is required
to reenter.

Note: Program written in Visual C++ .NET
---------------------------------------------------------------------------*/
#include <iostream>           // Needed for input/output stream
using namespace std;

void main()
{
  const MAX_CHARS = 1000;  // Max number of illegal characters to ignore

  char ErrorPrompt[] = "\nError in input: Press Enter key to continue";

  int N, InputNotGood;


  do
  {
    cout << "\nEnter an int => ";
     cin >> N;
     cout << endl;

     InputNotGood = !cin.good();     // check to see if input error occurred

     if ( InputNotGood )
     {
       cin.clear();                 //reset stream state to allow more input

       cin.ignore(MAX_CHARS, '\n');  //read and ignore up to MAX_CHARS
                                    //Stop when newline read.
       cout << ErrorPrompt;

       cin.ignore(MAX_CHARS, '\n');  //wait for user to press enter key
     }
   }
   while ( InputNotGood );

   cout << "\nN = " << N << endl;
}

/* ------------------------  program output -------------------------

Enter an int => t56

Error in input: Press Enter key to continue

Enter an int => e

Error in input: Press Enter key to continue

Enter an int => 5

N = 5                         */
```