C++ Input/Output: Streams

The basic data type for I/O in C++ is the stream. C++ incorporates a complex hierarchy of stream types. The most basic stream types are the standard input/output streams:

`istream cin`	built-in input stream variable; by default hooked to keyboard
`ostream cout`	built-in output stream variable; by default hooked to console

header file: <iostream>

C++ also supports all the input/output mechanisms that the C language included. However, C++ streams provide all the input/output capabilities of C, with substantial improvements.

We will exclusively use streams for input and output of data.

C++ Streams are Objects

The input and output streams, `cin` and `cout` are actually C++ objects. Briefly:

class:	a C++ construct that allows a collection of variables, constants, and functions to be grouped together logically under a single name
object:	a variable of a type that is a class (also often called an instance of the class)

For example, `istream` is actually a type name for a class. `cin` is the name of a variable of type `istream`.

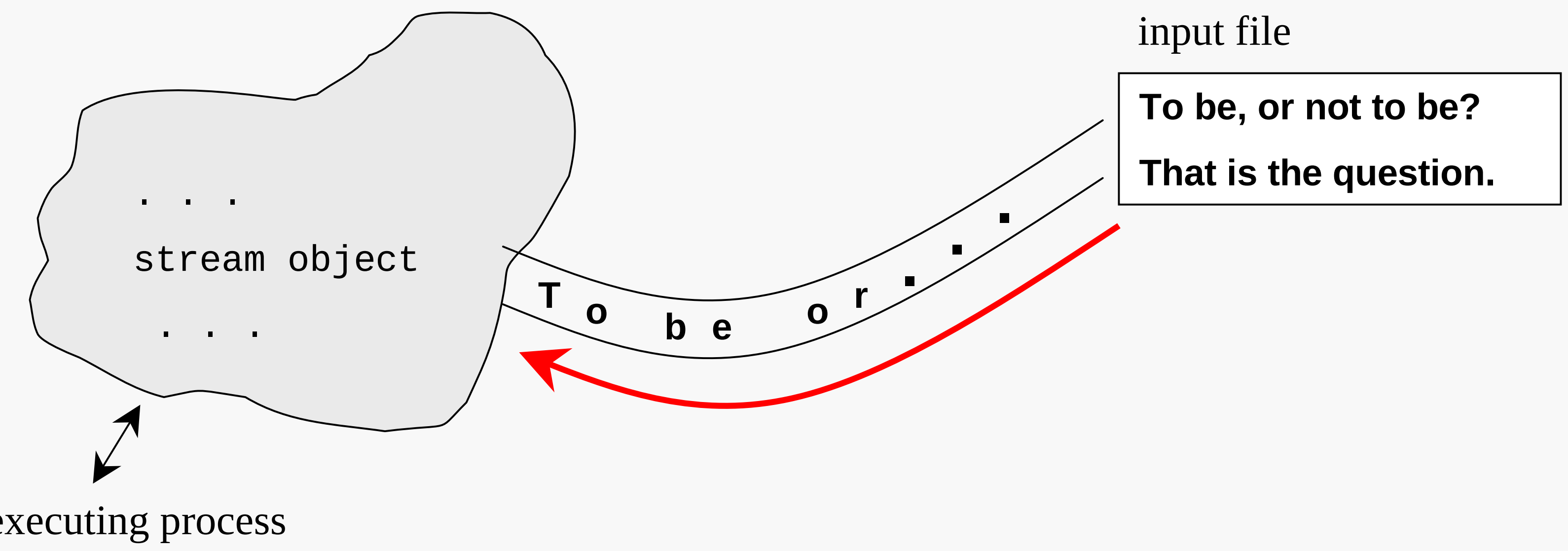
So, we would say that `cin` is an instance or an object of the class `istream`.

An instance of a class will usually have a number of associated functions (called member functions) that you can use to perform operations on that object or to obtain information about it. The following slides will present a few of the basic stream member functions, and show how to go about using member functions.

Classes are one of the fundamental ideas that separate C++ from C. In this course, we will explore the standard stream classes and the standard string class.

Conceptual Model of a Stream

A stream provides a connection between the process that initializes it and an object, such as a file, which may be viewed as a sequence of data. In the simplest view, a stream object is simply a serialized view of that other object. For example, for an input stream:



We think of data as flowing in the stream to the process, which can remove data from the stream as desired. The data in the stream cannot be lost by “flowing past” before the program has a chance to remove it.

The stream object provides the process with an “interface” to the data.

Output: the Insertion Operator

To get information out of a file or a program, we need to explicitly instruct the computer to output the desired information.

One way of accomplishing this in C++ is with the use of an output stream.

In order to use the standard I/O streams, we must have in our program the pre-compiler directive:

```
#include <iostream>
```

In order to do output to the screen, we merely use a statement like:

```
cout << " X = " << X;
```

Hint: the insertion operator (<<) points in the direction the data is flowing.

where X is the name of some variable or constant that we want to write to the screen.

Insertions to an output stream can be "chained" together as shown here. The left-most side <u>must</u> be the name of an output stream variable, such as `cout`.

Output Examples

Inserting the name of a variable or constant to a stream causes the value of that object to be written to the stream:

```
const string Label = "Pings echoed: 
";
int totalPings = 127;
cout << Label << totalPings << endl;
```

```
Pings echoed: 127
```

`endl` is a <u>manipulator</u>.

A <u>manipulator</u> is a C++ construct that is used to control the formatting of output and/or input values.

Manipulators can only be present in Input/Output statements. The `endl` manipulator causes a newline character to be output.

`endl` is defined in the <`iostream`> header file and can be used as long as the header file has been included.

No special formatting is supplied by default.

Alignment, line breaks, etc., must all be controlled by the programmer:

```
cout << "CANDLE" << endl;
cout << "STICK" << endl;
```

```
cout << "CANDLE";
cout << "STICK" << endl;
```

Input: the Extraction Operator

To get information into a file or a program, we need to explicitly instruct the computer to acquire the desired information.

One way of accomplishing this in C++ is with the use of an input stream.

As with the standard input stream, cout, the program must use the pre-compiler directive:

```
        #include <iostream>
```

In order to do output, we merely use a statement like:

```
        cin >> X;
```

Hint: the extraction operator (>>) points in the direction the data is flowing.

where `X` is the name of some variable that we want to store the value that will be read from the keyboard.

As with the insertion operator, extractions from an input stream can also be "chained". The left-most side must be the name of an input stream variable.

Input Examples

Assume the input stream `cin` contains the data:

```
12  17.3  -19
```

Then:

```
int    A, B;
double X;
cin >> A;  // A <--- 12
cin >> X;  // X <--- 17.3
cin >> B;  // B <--- -19
```

If we start each time with the same initial values in the stream:

```
int  A, B;
char C;
cin >> A;  // A <--- 12
cin >> B;  // B <--- 17
cin >> C;  // C <--- '.'
cin >> A;  // A <--- 3
```

```
int  A;
char B, C, D;
cin >> A;  // A <--- 12
cin >> B;  // B <--- '1'
cin >> C;  // C <--- '7'
```

The extraction operator is "smart enough" to consider the type of the target variable when it determines how much to read from the input stream.

`string` Input with Extraction

The extraction operator may be used to read characters into a string variable.

The extraction statement reads a whitespace-terminated string into the target string, ignoring any leading whitespace and not including the terminating whitespace character in the target string.

Assume the input stream `cin` contains the data:

```
Flintstone, Fred     718.23
```

Then:

```
string L, F;
double X;
cin >> L;  // L <--- "Flintstone,"
cin >> F;  // F <--- "Fred"
cin >> X;  // X <--- 718.23
```

The amount of storage allocated for the string variables will be adjusted as necessary to hold the number of characters read. (There is a limit on the number of characters a string variable can hold, but that limit is so large it is of no practical concern.)

Of course, it is often desirable to have more control over where the extraction stops.

Extraction Operator and Whitespace

In programming, common characters that do not produce a visible image on a page or in a file are referred to as whitespace.

The most common whitespace characters are:

Name	Code
newline	`\n`
tab	`\t`
blank	(space)
carriage return	`\r`
vertical tab	`\v`

By default, the extraction operator in C++ will ignore leading whitespace characters.

That is, the extraction operator will remove leading whitespace characters from the input stream and discard them.

What if we need to read and store whitespace characters? See the `get()` function later in the notes.

Details of an Extraction

Assume the input stream cin contains:

```
12  17.3  -19
```

The numbers are separated by some sort of whitespace, say by tabs.

Suppose that `X` is declared as an `int`, and the following statement is executed:

```
cin >> X;
```

The type of the targeted variable, `X` in this case, determines how the extraction is performed.

First, any leading whitespace characters are discarded.

Since an integer value is being read, the extraction will stop if a character that couldn't be part of an integer is found.

So, the digits '1' and '2' are extracted, and the next character is a tab, so the extraction stops and `X` gets the value 12.

The tab after the '2' is left in the input stream.

`ignore()` Member Function

There is also a way to remove and discard characters from an input stream:

```
        cin.ignore(N, ch);
```

means to skip (read and discard) up to `N` characters in the input stream, or
until the character `ch` has been read and discarded, whichever comes first. So:

```
        cin.ignore(80, '\n');
```

says to skip the next 80 input characters <u>or</u> to skip characters until a newline character is read, whichever comes first.

The ignore function can be used to skip a specific number of characters or halt whenever a given character occurs:

```
        cin.ignore(100, '\t');
```

means to skip the next 100 input characters, or until a tab character is read, or whichever comes first.

Interactive I/O

Prompts: users must be given a cue when and what they need to input:

```
const string AgePrompt = "Enter your Age: ";
cout << AgePrompt;
cin  >> UserAge;
```

The statements above allow the user to enter her/his age in response to the prompt.

Because of buffering of the I/O by the computer, it is possible that the prompt may not appear on a monitor before the program expects input to be entered.

To ensure output is sent to its destination immediately:

```
cout << AgePrompt << flush;
cin  >> UserAge;
```

The manipulator `flush` ensures that the prompt will appear on the display before the input is required.

The manipulator `endl` includes a implicit `flush`.

Streams for File I/O

C++ also provides stream types for reading from and writing to files stored on disk. For the most part, these operate in exactly the same way as the standard I/O streams, `cin` and `cout`.

For basic file I/O: `#include <fstream>`

There are no pre-defined file stream variables, so a programmer who needs to use file streams must declare file stream variables:

```
ifstream inFile;     // input file stream object
ofstream outFile;    // output file stream object
```

The types `ifstream` and `ofstream` are C++ stream classes designed to be connected to input or output files.

File stream objects have all the member functions and manipulators possessed by the standard streams, `cin` and `cout`.

Connecting Streams to Files

By default, a file stream is not connected to anything. In order to use a file stream the programmer must establish a connection between it and some file. This can be done in two ways.

You may use the `open()` member function associated with each stream object:

```
inFile.open("readme.data");
outFile.open("writeme.data");
```

This sets up the file streams to read data from a file called "readme.data" and write output to a file called "writeme.data".

For an input stream, if the specified file does not exist, it will not be created by the operating system, and the input stream variable will contain an error flag. This can be checked using the member function `fail()` discussed on a later slide.

For an output stream, if the specified file does not exist, it will be created by the operating system.

Connecting Streams to Files

You may also connect a file stream variable to a file when the stream variable is declared:

```
ifstream inFile("readme.data");
ofstream outFile("writeme.data");
```

This also sets up the file streams to read data from a file called "readme.data" and write output to a file called "writeme.data".

The only difference between this approach and using the `open()` function is compactness.

Warning: if you use a `string` constant (or variable) to store the file name, you must add a special conversion when connecting the stream:

```
string   inputFileName = "readme.data";
ifstream inFile(inputFileName.c_str());
```

`close()` Member Function

When a program is finished with a file, it must close the file using the `close( )` member function associated with each file stream variable:

```
        inStream.close( );
        outStream.close( );
```

(Including the file name is an error.)

Calling `close( )` notifies the operating system that your program is done with the file and that the system should flush any related buffers, update file security information, etc.

It is always best to close files explicitly, (even though by the C++ standard, files are closed automatically whenever the associated file stream variable goes out of scope [see the chapter on functions for a presentation of scope]).

Formatting Numeric Output

First of all you need to include the manipulator header file: `<iomanip>`

`setw( ):`

sets the field width (number of spaces in which the value is displayed).
`setw( )` takes one parameter, which must be an integer.

The `setw( )` setting applies to the next single value output only.

`setprecision( ):`

sets the precision, the number of digits shown after the decimal point.
`setprecision( )` also takes one parameter, which must be an integer.

The `setprecision( )` setting applies to all subsequent floating point values, until another `setprecision( )` is applied.

Floating Point Formatting

In addition, to activate the manipulator `setprecision( )` for your output stream, insert the following two manipulators once:

```
outStream << fixed << showpoint;
```

(Just use the name of your output stream variable.)

Omitting these manipulators will cause `setprecision( )` to fail, and will cause real values whose decimal part is zero to be printed without trailing zeroes regardless of `setprecision( )`.

Other useful manipulators:

```
bin
hex
octal
dec
scientific
```

Setting Justification

Justification

- Justification refers to the alignment of data within a horizontal field.
- The default justification in output fields is to the right, with padding occurring first (on the left).
- To reverse the default justification to the left:

```
cout << fixed << showpoint;
string empName = "Flintstone, Fred";
double Wage  = 8.43;
double Hours = 37.5;
cout << left;              //turn on left justification
cout << setw(20) << empName;
cout << right;             //turn on right justification
cout << setw(10) << setprecision(2) << Wage * Hours << endl;
```

This will produce the output:

```
012345678901234567890123456789

Flintstone, Fred        316.13
```

Setting Padding

Padding Output

- Padding refers to the character used to fill in the unused space in an output field.
- By default the pad character for justified output is the space (blank) character.
- This can be changed by using the `setfill()` manipulator:

```
int ID = 413225;
cout << "0123456789" << endl;
cout << setw(10) << ID << endl;
cout << setfill('0');   //pad with zeroes
cout << setw(10) << ID << endl;
cout << setfill(' ');   //reset padding to spaces
```

This will produce the output:

```
0123456789
    413225
0000413225
```

Example with Manipulators

```cpp
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

int main() {

   ofstream oFile("AreaData.out");
   oFile << fixed << showpoint;

   const double PI = 3.141592654;
   string figName = "Ellipse";
   int majorAxis = 10,
       minorAxis =  2;
   double Area = PI * majorAxis * minorAxis;

   oFile << setw(20) << "Area" << endl;
   oFile << left  << setw(10) << figName;
   oFile << right << setw(10) << setprecision(4)
         << Area << endl;

   oFile.close();
   return 0;
}
```

```
                Area

Ellipse      62.8319
```

Input Failure

When you attempt to read a value from an input stream, the extract operator or other input function takes into account the type of the variable into which the value is to be stored. If there is a default conversion between the type of the data in the stream and the type of the target variable, then that is applied and all is well.

What happens if the next data in the input stream is not compatible with the target variable?

In that case, the input operation fails.

The effect on the target variable is compiler-dependent. With Visual C++, the target variable is generally not modified (`string` variables are an exception).

The stream variable sets an internal flag indicating it is in a "fail state". Subsequent attempts to read from the stream will automatically fail (see `clear()` later in these notes).

In consequence, it is vital that C++ programmers design input code to reflect the formatting of the input data. Programming to handle general, unstructured input data is extremely difficult.

Input Examples

Consider the following input code fragments and associated input streams:

```
int    iA, iB;
double dX, dY;
char   cC;
string sS;

// input code                       stream data (space separated)
In >> iA >> iB;                     // 17   x   42

In >> dX >> dY;                     // 73   .2

In >> iA >> cC >> iB;               // 73   .2

In >> cC >> iX >> sS;               // 42   23bravoxray

In >> iA >> cC >> sS;               // 42   23bravoxray
```

Reading to Input Failure

When you attempt to extract a value from an input stream, the stream variable returns an indication of success (true) or failure (false). You can use that to check whether the input operation has failed for some reason.

A `while` loop is used to extract data from the input stream until it fails.

Note well: the design here places a test to determine whether the read attempt succeeded between each attempt to read data and each attempt to process data. Any other approach would be logically incorrect.

Design Logic

```
Try to read data.  (Often called the priming read.)
While the last attempt to read data succeeded do
        Process the last data that was read.
        Try to read data.
Endwhile
```

This is one of the most common patterns in programming.

Failure-Controlled Input Example

```
. . .
const int MINPERHOUR = 60;
int Time;       // time value in total minutes
int Hours,      // HH field of Time
    Minutes;    // MM field of Time

int numTimes  = 0;  // number of Time values read
int totalTime = 0;  // sum of all Time values

In >> Time;
while ( In ) {
   numTimes++;
   totalTime = totalTime + Time;
   Hours   = Time / MINPERHOUR;
   Minutes = Time % MINPERHOUR;
   Out << setw(5) << numTimes << "|";
   Out << setw(5) << Hours << ":";
   Out << setw(2) << setfill('0')
       << Minutes << setfill(' ') << endl;

   In >> Time;
}

Out << "Total minutes: " << setw(5) << totalTime
    << endl;
. . .
```

Input:

```
217
 49
110
302
 91
109
198
```

Output:

```
    1|    3:37
    2|    0:49
    3|    1:50
    4|    5:02
    5|    1:31
    6|    1:49
    7|    3:18
Total minutes:  1076
```

Failure-Controlled Input Example

The program given on the previous slide will terminate gracefully if the input file contains an error that causes the input operation to fail:

Input:

```
217
 49
110
xxx
 91
109
198
```

Output:

```
     1|    3:37
     2|    0:49
     3|    1:50
Total minutes:  376
```

Trace the execution…

The input-failure logic used in the sample code has more than one virtue:

- it will terminate automatically at the end of a correctly-formatted input file.
- it will terminate automatically if an input failure occurs while reading an incorrectly-formatted input file, acting as if that point were in fact the end of the input file.

Of course, it would be nice if an error message were also generated above…

Incorrect Design

```
. . .
// NO priming read
while ( In ) {
      In >> Time;     // Read at beginning of loop

      numTimes++;     // . . . then process data.

      totalTime = totalTime + Time;
      Hours = Time / MINPERHOUR;
      Minutes = Time % MINPERHOUR;

      Out << setw(5) << numTimes << "|";
      Out << setw(5) << Hours << ":";
      Out << setw(2) << setfill('0')
          << Minutes << setfill(' ') << endl;
   }
. . .
```

Input:

```
217
 49
110
302
 91
109
198
```

Output:

```
    1|    3:37
    2|    0:49
    3|    1:50
    4|    5:02
    5|    1:31
    6|    1:49
    7|    3:18
    8|    3:18
Total minutes:  1274
```

This is a classic error!

Note that the last time value is output twice. That's typical of the results when this design error is made… watch for it.

String Input: `getline( )`

The `getline( )` standard library function provides a simple way to read character input into a string variable, controlling the “stop” character.

Suppose we have the following input file:

```
Fred Flintstone         Laborer         13301
Barney Rubble           Laborer         43583
```

There is a single tab after the employee name, another single tab after the job title, and a newline after the ID number.

Assuming `iFile` is connected to the input file above, the statement

```
getline(iFile, String1);
```

would result in `String1` having the value:

```
"Fred Flintstone   Laborer          13301"
```

More on `getline( )`

As used on the previous slide, `getline( )` takes two parameters. The first specifies an input stream and the second a string variable.

Called in this manner, `getline( )` reads from the current position in the input stream until a newline character is found.

Leading whitespace is included in the target string.

The newline character is removed from the input stream, but not included in the target string.

It is also possible to call `getline( )` with three parameters. The first two are as described above. The third parameter specifies the "stop" character; i.e., the character at which `getline( )` will stop reading from the input stream.

By selecting an appropriate stop charcter, the `getline()`function can be used to read text that is formatted using known delimiters. The example program on the following slides illustrates how this can be done with the input file specified on the preceding slide.

String Input Example

```
#include <fstream>                 // file streams
#include <iostream>                // standard streams
#include <string>                  // string variable support
#include <climits>
using namespace std;               // using standard library

int main() {

   string EmployeeName, JobTitle;   // strings for name and title
   int    EmployeeID;               // int for id number

   ifstream iFile("Employees.data");
                                       // Priming read:
   getline(iFile, EmployeeName, '\t'); //    read to first tab
   getline(iFile, JobTitle, '\t');     //    read to next tab
   iFile >> EmployeeID;                //    extract id number
   iFile.ignore(INT_MAX, '\n');        //    skip to start of next line
. . .
```

String Input Example

```
. . .
   while (iFile) {                     // read to input failure

      cout << "Next employee: " << endl;     // print record header
      cout << EmployeeName << endl           // name on one line
           << JobTitle                       // title and id number
           << EmployeeID   << endl << endl;  //    on another line

      getline(iFile, EmployeeName, '\t');    // repeat priming read
      getline(iFile, JobTitle, '\t');        //    logic
      iFile >> EmployeeID;
      iFile.ignore(INT_MAX, '\n');
   }

   iFile.close();                   // close input file
   return 0;
}
```

fail( ) Member Function

`fail( )` provides a way to check the status of the last operation on the input stream.

`fail( )` returns true if the last operation failed and returns false if the operation was successful.

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

void main( ) {
   ifstream inStream;
   inStream.open("infile.dat");

   if  ( inStream.fail() ) {
      cout << "File not found.  Please try again." ;
      return;
   }
// . . . omitted statements doing something useful . . .
   inStream.close();
}
```

`clear()` Member Function

`clear( )` provides a way to reset the status flags of an input stream, after an input failure has occurred.

Note: closing and re-opening the stream does NOT clear its status flags.

Consider designing a program to read an input file containing statistics for a baseball player, as shown below, and to produce a summary:

```
Hits   At-Bats
   3         4
   2         3
   1         3
   2         4
Hammerin' Hokie
```

Unless we know exactly how many lines of batting data are going to be given, we must use an input-failure loop to read the batting data. But then the input stream will be in a fail state, and we still need to read the player's name.

We can use `clear()` to recover from the input failure and continue reading input.

Using `clear()`

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

int main() {

   ifstream In("Hitting.data");

   string playerName;            // player's name
   int Hits, atBats;             // # of hits and at-bats in current game
   int numGames    = 0;          // # of games reported
   int totalHits   = 0,          // total # of hits in all games
       totalAtBats = 0;          // total # of at-bats in all games

   In.ignore(INT_MAX, '\n');     // skip over header line

   In >> Hits >> atBats;         // try to read 1st game data
   while ( In ) {
      totalHits = totalHits + Hits;         // update running totals
      totalAtBats = totalAtBats + atBats;
      numGames++;                           // count this game
      In >> Hits >> atBats;                 // try to read next game data
   }
. . .
```

Using `clear()`

```
. . .
   // Recover from the read failure at the end of the
   // batting data:
   In.clear();
   // Read the player's name:
   getline(In, playerName);

   // Calculate the batting average:
   double battingAverage = double(totalHits) / totalAtBats;

   // Write the results:
   cout << fixed << showpoint;
   cout << playerName << " is batting "
        << setprecision(3) << battingAverage
        << " in " << numGames << " games."
        << endl;

   In.close();
   return 0;
}
```

`eof()` Member Function

Every file ends with a special character, called the end-of-file mark.

`eof()` is a boolean function that returns true if the last input operation attempted to read the end-of-file mark, and returns false otherwise.

The program on slide 4.24 could be modified as follows to use `eof()` to generate an error message if an input failure occurred in the loop:

```
. . .
Out << "Total minutes: " << setw(5) << totalTime
    << endl;

if ( !In.eof() ) {
      Out << endl
          << "An error occurred while reading the file." << endl
          << "Please check the input file." << endl;
   }
. . .
```

In general, reading until input failure is safer than reading until the end-of-file mark is reached. DO NOT use `eof()` as a substitute for the input-failure logic covered earlier.

`get()` Member Function

The input stream object `cin` has a member function named `get( )` which returns the next single character in the stream, whether it is whitespace or not.

To call a member function of an object, state the name of the object, followed by a period, followed by the function call:

```
   cin.get(someChar);    // where someChar is a char variable
```

This call to the `get( )` function will remove the next character from the stream `cin` and place it in the variable `someChar`.

So to read all three characters (from the previous slide), we could have:

```
   cin.get(ch1);        // read ‘A’
   cin.get(someChar);   // read the space
   cin.get(ch2);        // read ‘M’
```

Other Useful Member Functions

The `istream` class provides many additional member functions. Here are two that are often useful:

`peek()` provides a way to examine the next character in the input stream, without removing it from the stream.

```
. . .
char nextCharacter;
nextCharacter = In.peek();
. . .
```

`putback()` provides a way to return the last character read to the input stream.

```
. . .
const char PUTMEBACK = '?';
char nextCharacter;
In.get(nextCharacter);
if ( nextCharacter == PUTMEBACK ) {
   In.putback(nextCharacter);
}
. . .
```