Parsing Input Data

Streams

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

Standard Input Streams **header file: <iostream>**

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

Note: cin and cout are predefined variables, not types.

File Stream Types **header file: <fstream>**

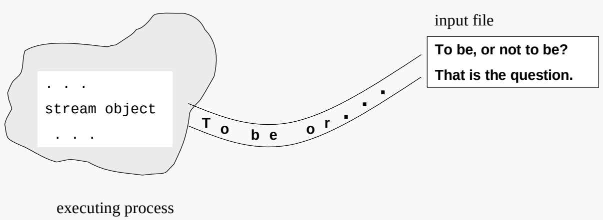
- `ifstream` hooked to desired input file by use of `open()` member function
- `ofstream` hooked to desired output file similarly

String Stream Types **header file: <stringstream>**

- `istringstream` hooked via constructor to a string object for input
- `ostringstream` hooked via constructor to a string object for output

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.



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.

Associating a File Stream with a File

Two basic methods:

object constructor:

```
ifstream In("infoo.txt");
ofstream Out("outfoo.txt");
```

`open()`:

```
ifstream In;
In.open("infoo.txt");
ofstream Out;
Out.open("outfoo.txt");
```

File must (normally) be in current directory.

If named input file is not found, the stream is not properly initialized.

If named output file is not found, an empty file of that name is created.

If named output file is found, it is opened and its contents deleted (truncated).

When finished with a file, input or output, the user should invoke the close() member function to signal that fact to the OS:

Out.close();

That's right, no file name is used.

Never, ever, call close() on cin or cout.

Basic Stream Input

Because the various stream types are related (via inheritance, which comes later), there is a common set of operations for input and output that all support. In the discussion below, `In` can be any type of input stream object and `Out` any type of output stream object.

Input via extraction: `In >> TargetVariable;`

- `>>` is the extraction operator
- left hand side must be an input stream variable
- right hand side must be a variable of a built-in type (pending overloading later)
- the operation attempts to extract the first complete “object” from the stream that matches the target variable in type; some automatic conversions (such as int to double) are supported
- leading whitespace is automatically ignored (I.e., extracted and discarded)
- in general, the type of the target variable should conform to the type of data that will occur next in the input stream
- extractions may be chained, as:

```
In >> var1 >> var2 >> var3 >> . . .
```


Basic Stream Output

Output via insertion: `Out << SourceVariable;`

- `<<` is the insertion operator
- left hand side must be an output stream variable
- right hand side must be a variable of a built-in type (pending overloading later)
- the operation attempts to write to the output stream a sequence of characters (keep it simple for now) that represents the value of the source variable; some automatic formatting rules are supported
- whitespace is not automatically inserted between inserted values
- user may also use manipulators to control the formatting precisely
- insertions may be chained, as:

```
Out << var1 << var2 << var3 << . . .
```


Reading Single Characters: get()

Input stream objects have a member function named `get( )` which returns the next single character in the stream, whether it is whitespace or not.

```
char someChar;
In.get(someChar);
```

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

If we had a stream containing “A M” (one space between A and M) we could read all three characters by;

```
char ch1, ch2, ch3;
In >> ch1;         // read ‘A’
In.get(ch2);       // read the space
In >> ch3;         // read ‘M’
```

We could also have used the `get()` function to read all three characters.

Skipping and Discarding Characters: ignore()

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

```
In.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:

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

says to skip the next 80 input characters or 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:

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

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

Skipping and Discarding Characters: ignore()

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

```
In.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:

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

says to skip the next 80 input characters or 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:

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

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

Numeric Output Manipulators

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.

To activate manipulators for floating point output, include:

```
Out.setf(ios::fixed, ios::floatfield);
Out.setf(ios::showpoint);
```

Omitting the `fixed`, `floatfield` and `showpoint` flags will cause `setprecision( )` to fail, and will cause integer values to be printed without trailing zeroes regardless of `setprecision( )`.

Padding and Justification Manipulators

Padding Output

- By default the pad character for justified output is the space (blank) character.
- This can be changed by using the `fill()` manipulator:

```
Out.fill('0');   //pad with zeroes
Out << setw(9) << StudentID;
Out.fill(' ');   //reset padding
```

Left Justification

- 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:

```
Out.setf(ios::left);   //turn on left justification
// insert left justified output statements here
Out.unsetf(ios::left); //turn off left justification
```


Reading to Input Failure

When you attempt to extract a value from the input stream, the stream variable returns an indication of success (true) or failure (false). You can use that to check for when you've reached the end of the file from which you're reading data, or if the input operation has failed for some other reason.

A while loop may be used to extract data from the input stream, stopping automatically when an input failure occurs.

Note well: a preliminary or priming read is used before the while loop. Failure to do that will almost certainly lead to incorrect performance (see slide 14).

```
Now is the¶
time for¶
all good men¶
to come to the¶
aid of their party!¶§
```

¶	represents the return char
§	represents the end of file char

Failure-Controlled Input Example


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

void main( ) {
   int anInt;
   ifstream inStream;
   ofstream outStream;
   inStream.open("infile.dat");
   outStream.open("outfile.dat");

   inStream >> anInt;                // priming read before loop

   while (inStream) {                // check for read failure
      outStream << anInt << endl;    // print value
      inStream >> anInt;             // read next value at end of
   }                                 //    the loop body

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

It is important to understand the logic of this program. Reading to input failure is often necessary and alternative logical designs are likely to be incorrect.

Failure-Controlled Input Example

The program given on the previous slide will produce the output file shown below from the input file shown below:

infile.dat

```
171      32      41      17§
```

outfile.dat

```
171¶
32¶
41¶
17¶§
```

. . . and it will produce the output file shown below from the input file shown below:

infile.dat

```
171      32      Fred      17§
```

outfile.dat

```
171¶
32¶§
```

At this point, an integer is expected, and the next data is not a valid digit or '+' or '-'. An input failure occurs and the stream fails.

Incorrect Failure-Controlled Input


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

void main( ) {
   int anInt;
   ifstream inStream;
   ofstream outStream;
   inStream.open("infile.dat");
   outStream.open("outfile.dat");
   // no priming read before loop
   while (inStream) {                // check for read failure
      inStream >> anInt;             // read next value at start
                                     //    of the loop body
      outStream << anInt << endl;    // print value
   }
   inStream.close( );
   outStream.close( );
}
```

```
171      32      41      17§
```

→

```
171¶
32¶
41¶
17¶
17¶§
```

This program will not produce correct output. Logically, the problem is that the last input operation is not followed immediately by a test for success/failure.

Detecting end-of-file: eof( )

The end of a file is marked by a special character, called the end-of-file or EOF marker.

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

The loop test in the program on the previous slide could be modified as follows to use `eof( )`:

```
   inStream >> anInt;

   while (!inStream.eof()) {          // check for eof()
          outStream << anInt;         // print value
          inStream >> anInt;          // read next value
   }
```

This while loop will terminate when `eof( )` returns false.

In general, reading until input failure is safer than the technique illustrated here. The code shown above will <u>not</u> terminate gracefully if an input failure occurs in the middle of the input file.

Look-ahead parsing: peek( )

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

For example, the following code skips whitespace characters in the input stream:

```
char ch;
ch = inFile.peek();  // peek at first character

// while the first character is a space, tab or newline
while ( (ch == ' '  || ch == '\t' || ch == '\n') && (inFile) ) {

   inFile.get(ch);          // remove it from the stream

   ch = inFile.peek();      // peek at the (new) first char
}
```


Changing your mind: putback( )

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

For example, the following code also skips whitespace characters in the input stream:

```
char ch;
inFile.get(ch);     // remove first character from stream

// while you just got a space, tab or newline
while ( (ch == ' '  || ch == '\t' || ch == '\n') && (inFile) ) {

   inFile.get(ch);  // remove next character from stream
}

inFile.putback(ch);   // last character read was
                      // not whitespace, so put it back
```


Checking for Stream Failure: fail( )

`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("infile.dat");

   if ( inStream.fail() ) {      // !In will also work
      cout << "File Not Found";
      return;
   }

   // . . . now do interesting stuff . . .
}
```


Recovering from Stream Failure: clear( )

If an input stream goes into a fail state, it remains in that state unless it is explicitly reset. Even closing and re-opening the file will not work.

`clear( )` provides a way to restore a failed stream to use.

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

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

   // priming read
   while ( inStream ) {
      // process last data read
      // try to read next data
   }

   inStream.clear();

   // . . . now do more stuff with inStream . . .
}
```


Working with Character Strings

The C++ language provides three ways to deal with sequences of characters:

- string literals (constants) such as: `“Hello, world”`
- C-style arrays of `char` such as: `char myCharArray[100];`
- string objects such as: `string myStringObject;`

From a modern perspective, the addition of the `string` type to the C++ language renders the use of `char` arrays for variable character data obsolete.

String objects are simpler to use because they adjust to the size of the data stored and eliminate the problems associated with the array dimension.

String objects provide a robust library of member functions to manipulate character data.

String objects are type-safe, and may be used for the return value from a function, unlike an array.

The following notes discuss parsing with string objects. For a more general overview of string objects, see the Chapter 12 on String Objects in the CS 1044 notes (online).

String Objects

`string` type may be declared and optionally initialized as:

header file: <string>

```
string Greetings;
string Greetings2(“Hello, world!”);   // constructor syntax
string Greetings3 = “Hello, world!”;  // initialization syntax
```

`string` objects may be assigned using `=`, and compared using `==`, `>`, `<`, etc.

`string` objects do NOT store their data as a C-style null-terminated `char` array.

The limit on the number of characters a `string` object can store can be found using the member function `capacity()`:

```
cout << Greetings2.capacity() << endl;
```

Prints 31

However, the capacity will increase automatically as needed:

```
Greetings2 = "Everything should be made as simple as possible";
cout << Greetings2.capacity() << endl;
```

Prints 63

String Output

A string variable may be printed by inserting it to an output stream, just as with any simple variable:

```
cout << Greetings3 << endl;
```

Just as with string literals, no whitespace padding is provided automatically, so:

```
cout << Greetings3 << “It’s a wonderful day!”;
```

would print:

```
Hello, world!It’s a wonderful day!
```

as opposed to:

```
cout << Greetings3 << “  ” << “It’s a wonderful day!”;
```


String Input: extraction

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

```
string Greetings;
In >> Greetings;
```

The extraction statement reads a whitespace-terminated string into the target `string` (`Greetings` in this case), ignoring any leading whitespace and not including the terminating whitespace character in the target `string`.

The amount of storage allocated for the variable `Greetings` 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 concern.)

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

Delimited 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:

String1.dat

```
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 statements

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

Whereas, the statement

iFile >> String1;

would have stored "Fred" in String1.

would result in String1 having the value:

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


Delimited Input: 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 is a `char`, which specifies the "stop" character; i.e., the character at which `getline( )` will stop reading from the input stream.

By selecting an appropriate stop character, 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 previously.

Delimited Input Example

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

void main() {

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

   string fName = "String1.dat";
   ifstream iFile( fName.c_str() );


   if ( iFile.fail() ) {
      cout << "File not found: " << fName << endl;;
      return;
   }
                                            // 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(80, '\n');                  //    skip to start of next line
```

Member function c_str() returns a C-style string, which is what open() requires.

See later slide for better error handling.

Delimited Input Example

```
   while (iFile) {                                  // read to 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(80, '\n');
   }

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

This program takes advantage of the formatting of the input file to treat each input line as a collection of logically distinct entities (a name, a job title, and an id number). That is generally more useful than simply grabbing a whole line of input at once.

Improved Error Handling

The way the previous program responds to a missing input file can be improved:

```
// . . .
   string fName = "String1.dat";
   ifstream iFile(fName.c_str());
   while ( iFile.fail() ) {

      iFile.clear();

      cout << "File not found:  " << fName << endl;
      cout << "Please enter new name: ";

      getline(cin, fName);

      cin.ignore(1, '\n');

      iFile.open(fName.c_str());
   }
// . . .
```

Clear the input stream following failure.

Prompt user for new file name.

Read the file name (until a newline is found). Now it gets ugly. The user has to press Return twice. Once to flush the keyboard buffer and once to satisfy getline(). That leaves an extra newline in the input stream.

Get rid of the second newline.

Try to open input file again.

Input StringStream Objects

C++ also provides input streams that may be hooked to `string` objects:

```
string Greetings("Hello, world!");
istringstream In(Greetings);
```

header file: <sstring>

`istringstream` objects may be used to parse the contents of `string` objects in much the same way that `ifstream` objects may be used with files:

```
In >> Word1 >> Word2;
cout << setw(3) << Word1.length() << ":" << Word1 << endl
     << setw(3) << Word2.length() << ":" << Word2 << endl;
```

will print:

```
6:Hello,
6:world!
```

That's the same behavior as if we were extracting from an `istream` or an `ifstream`.

There are times when it's easiest to grab an entire block of characters into a `string` object and then parse them with an `istringstream`; for one thing this allows you to back up as far as you like in the `string`.

StringStream Example


```
#include <fstream>                          // file streams
#include <iostream>                         // standard streams
#include <sstream>                          // string stream support
#include <string>                           // string variable support

using namespace std;                        // using standard library

void main() {

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

   string fName = "String.dat";
   ifstream iFile(fName.c_str());
   while ( iFile.fail() ) {
      iFile.clear();
      cout << "File not found:  " << fName << endl;
      cout << "Please enter new name: ";
      getline(cin, fName);
      cin.ignore(1, '\n');
      iFile.open(fName.c_str());
   }
```


StringStream Example


```
   getline(iFile, FullLine);  // read first line into a string

   while (iFile) {

      istringstream In(FullLine);

      getline(In, EmployeeName, '\t');
      getline(In, JobTitle, '\t');
      In >> EmployeeID;

      cout << "Next employee: " << endl;
      cout << EmployeeName << endl
           << JobTitle     << "     "
           << EmployeeID   << endl << endl;

      getline(iFile, FullLine);
   }

   iFile.close();
}
```

Associate an istringstream with FullLine.

Parse FullLine for the Name, Title and ID. Note that the operations are identical to those for an ifstream.

What's the advantage? Not much, here.

However, with this approach the contents of FullLine could be searched and/or modified with the usual string functions, in addition to being parsed.

At the least, stringstreams are a handy tool.

Output StringStream Objects

C++ also provides output streams that may be hooked to `string` objects:

header file: <sstring>

```
string Greetings;
ostringstream Out(Greetings);
```

`ostringstream` objects may be used to write the contents of `string` objects in much the same way that `ofstream` objects may be used with files:

```
cout << "Please enter your name: ";
string UserName;
cin >> UserName;                    // assume user enters Fred
Out << "Hello, " << UserName << endl;
```

`Greetings` will now contain: `"Hello, Fred"`

Moreover, you can even use output manipulators with `ostringstream` objects.

`ostringstream` objects are primarily useful for assembling complex output before committing it to file or the screen.