I/O streams

Michael Nowak

Texas A&M University

Acknowledgement: Lecture slides based on those created by Bjarne Stroustrup for use with his textbook

Notes

Overview

Notes

Overview

Notes

Introduction

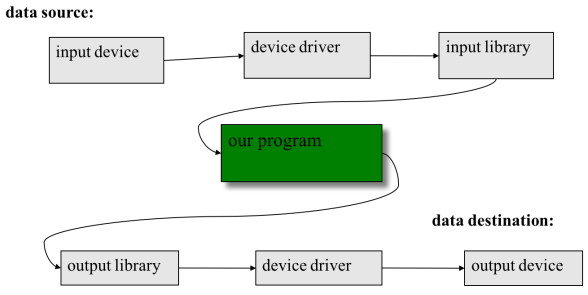
- The C++ language does not deal directly with input and output
- The C++ language s `standard library` provides a family of `types` that provide the I/O facilities
  - For instance, we have had to `#include<iostream>` to use the `objects`
    - `std::cin` to read from `standard input`
    - `std::cout` to write to `standard output`
  - For most programs, limiting I/O to the console window is insufficient
- The `standard library` provides different kinds of I/O types to support different kinds of I/O processing

Notes

Overview

Notes

Input & output



Notes

Overview

Notes

The stream model

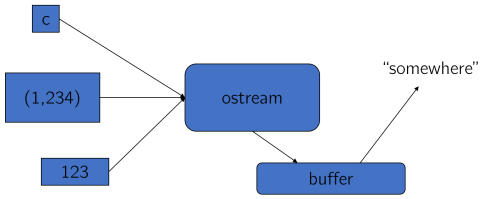
- A `stream` is a programming language construct that provides you with a character-based interface to I/O devices
- A common metaphor is a "stream of water": data is provided or consumed in a single-pass
  - Character data provided to a program from standard input flows through the `input stream` precisely once
  - Character data sent to standard output flows through the output stream to the console window precisely once
  - In both cases, the `stream` is nothing but a series of characters; its serial nature means that it can be traversed only once
  - You may have heard of buffered I/O; a `stream buffer` houses a fixed amount of extracted stream data

Notes

Overview

Notes

ostreams



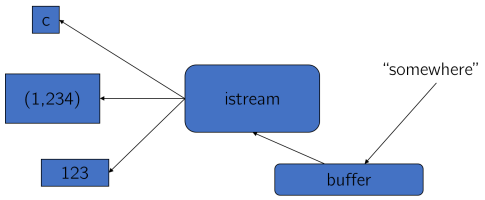
- An `ostream` turns values of various types into character sequences
- sends those characters somewhere
- `std::cout` is an `ostream` typed object that provides characters sequences to standard output

Notes

Overview

Notes

istreams



- An `istream` turns character sequences into values of various types
- gets those characters from somewhere
- `std::cin` is an `istream` typed object that consumes character sequences from standard input

Notes

Overview

Notes

Reading and writing

- Reading and writing of typed entities
  - `<<` (output) and `>>` (input) plus other operations
  - Type safe
  - Formatted
  - Typically stored (entered, printed, etc.) as text
    - But not necessarily (e.g., binary streams)

Notes

Reading from character streams

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using an `istream` in our program
  - We could read the input into an `std::string`
  - We could read the number `3`, followed by the character `*`, etc.
- It is completely up to us what type we would like to convert the characters into

  (as long as the character sequence is valid for the desired type)

Notes

Overview

Notes

The I/O classes

Header	Type
`iostream`	`istream` reads from a `stream`
`iostream`	`ostream` writes to a `stream`
`iostream`	`iostream` reads and writes a `stream`
`fstream`	`ifstream` reads from a `file`
`fstream`	`ofstream` writes a `file`
`fstream`	`fstream` reads and writes a `file`
`sstream`	`istringstream` reads from a `string`
`sstream`	`ostringstream` writes a string
`sstream`	`stringstream` reads and writes a string

- The above classes are provided to us by the standard library and allow for different kinds of I/O processing
  - To support languages that use wide characters, the library also provides a set of types and objects for `wchar_t` data

Notes

Overview

Notes

Relationship among I/O types

- A stream can be attached to any I/O or storage device
- The standard library lets us ignore the differences among different types of streams through the use of common operations
  - We can use `>>` to read data irrespective of whether we re reading from the console window, a disk file, or a string
  - Likewise, we can use `<<` to write data irrespective of whether we re writing to the console window, a disk file, or a string

Notes

Relationship among I/O types

- The standard library lets us ignore the differences among different types of streams by using `inheritance`
  - `ifstream` and `istringstream` `inherit` functionality from `istream`
  - This means that the features of `istream` are made available to `ifstream` and `istringstream`
  - So we can use objects of these types in the same manner that we ve used `std::cin`
  - Similarly, `ofstream` and `ostringstream` inherit from `ostream`
  - So we can use objects of these types in the same manner that we ve used `std::cout`

Notes

Overview

Notes

Files

- We turn our computers off and on
  - The contents of main memory are `volatile`
- We like to keep our data
  - We keep what we want to preserve on disks and similar permanent storage
- A file is a sequence of bytes stored in permanent storage
  - A file has a name
  - The data on a file has a format
- We can read/write a file if we know its name and format

Notes

A file

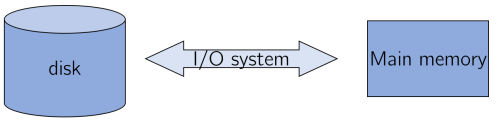
- At the fundamental level, a file is a sequence of bytes numbered from 0 upwards
- Other notions can be supplied by programs that interpret a "file format"
  - For example, the 6 `bytes` (`char`acters) "123.45" might be interpreted as the floating-point number 123.45

Notes

Overview

Notes

General model



iostreams

Files
(sequences of bytes)

Objects
(of various types)

Notes

Overview

Notes

General process

- To read a file, we must:
  - know its name
  - open it for reading
  - ensure that it opened successfully
  - read it
  - close it
- To write a file, we must:
  - know its name
  - open it for writing
    - or create a new file with that name
  - ensure that it opened successfully
  - write to it
  - close it

Notes

Overview

Notes

File streams

Header	Type
`fstream`	`ifstream` reads from a `file`
`fstream`	`ofstream` writes a `file`
`fstream`	`fstream` reads and writes a `file`

- These types provide the same operations as those we have used on with `std::cin` and `std::cout`

Notes

Overview

Notes

Using file stream objects

- When we want to read or write a file, we declare a `fstream` object and initialize it with the file name we d like to open
  - When we initialize the `fstream` object with a file name, that file is opened implicitly
    - When an `fstream` object is created, open is called automatically
  - `// construct an ifstream and open ifile`
    `ifstream in{ifile};`
  - `// construct an ofstream and open ofile`
    `ofstream out{ofile};`
  - When the `fstream` object s lifetime ends, the `fstream` will implicitly close the file
    - When an `fstream` object is destroyed, close is called automatically

Notes

Opening a file for reading

```
// ...

cout << "Please␣enter␣input␣file␣name:␣";
string iname;
cin >> iname;

ifstream ist {iname};
// ifstream is "aninput stream from a "file
// defining an ifstream with a name string
// opens the file of that name for reading

if (!ist) throw runtime_error("'cant␣open␣input␣file");

// ...
```

Notes

Opening a file for writing (discard contents)

```
// ...

cout << "Please␣enter␣name␣of␣output␣file:␣";
string oname;
cin >> oname;

ofstream ofs {oname};
// ofstream is an "output stream from a "file
// defining an ofstream with a name string
// opens the file with that name for writing
// the contents of the file are discarded

if (!ofs) throw runtime_error("'cant␣open␣output␣file");

// ...
```

Notes

Opening a file for writing (append to existing contents)

```
// ...

cout << "Please␣enter␣name␣of␣output␣file:␣";
string oname;
cin >> oname;

ofstream ofs {oname, ofstream::app};
// ofstream is an "output stream from a "file
// defining an ofstream with a name string
// opens the file with that name for writing
// the contents of the file are preserved

if (!ofs) throw runtime_error("'cant␣open␣output␣file");

// ...
```

Notes

Reading a file

- Suppose a file contains a sequence of pairs representing hours and temperature readings
  ```
  0 60.7
  1 60.6
  2 60.3
  3 59.22
  ```
- The hours are in the range

$$0, \ldots, 23$$

- No further format is assumed
- Termination
  - Reaching the end of file terminates the read
  - Anything unexpected in the file terminates the read

Notes

Reading a file

```
vector<int> hours;
vector<double> temps;

std::string iname = "temperatures.dat"
ifstream ist {iname};
if (!ist) throw runtime_error("'cant␣open␣input␣file");

int hour;
double temperature;
while (ist >> hour >> temperature)   // read
{
    // check
    if (hour < 0 || 23 < hour) error("hour␣out␣of␣range");
    hours.push_back(hour); // store
    temps.push_back(temperature); // store
}
```

Notes

Overview

Notes

I/O error handling

- Sources of errors:
  - Human mistakes
  - Files that fail to meet specifications
  - Specifications that fail to match reality
  - Programmer errors
  - etc.
- Some errors will be recoverable; others will be beyond the scope of a program to correct
- The I/O stream types define flags and functions that allow us to interrogate and manipulate the condition state of a stream
  - We can use a stream as a condition, e.x., `if (cin)`, to ask whether that stream is valid
  - If the condition evaluates `false`, we know we have a situation, but we re not sure why the stream is invalid (just yet)

Notes

Overview

Notes

Stream state flags

- The I/O stream types each provide a collection of bits (typed `iostate`) that are used to convey information about the state of a stream; different bit pattern are used to express different kinds of I/O conditions

Flag	Meaning
`goodbit`	Set when the stream is not in an error state
`badbit`	Set when an unrecoverable failure has occurred
`failbit`	Set when a recoverable error has occurred
`eofbit`	Set when the stream has hit end-of-file

Notes

Stream state flags

- Each I/O stream type has an `rdstate` function that returns an `iostate` value corresponding to the current state of a stream
- We could check whether the `failbit` is set for `cin` by writing:

```
if ((cin.rdstate() & std::ios::failbit ) != 0)
   /* failbit set, do something to recover */
```

- Similarly, we could check whether the `failbit` is set for an `ifstream` named `ifs` by writing:

```
if ((ifs.rdstate() & std::ifstream::failbit ) != 0)
   /* failbit set, do something to recover */
```

- Lucky for us, there is an easier way to check the current state of a respective stream

Notes

Overview

Notes

Stream state functions

- The I/O stream types have functions that can be used to interrogate the state of a stream

	Flag			
Function	goodbit	badbit	failbit	eofbit
good()	✓			
bad()		✓		
fail()		✓	✓	
eof()				✓

Notes

Overview

Notes

Validating input with stream state

```
string prompt = "Enter an integer: ";
cout << prompt;

int i;
cin >> i;

while (!cin.good())    // goodbit NOT set
{
    if (cin.bad()) { // badbit set
        /* do something */
    } else if (cin.eof()) { // eofbit set
        /* do something else */
    } else { // failbit set
        cout << "Invalid input!" << endl;
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << prompt;
        cin >> i;
    }
}
/* do something with valid input */
```

Notes

Validating input with stream state

- `cin.clear();`
  - Resets all conditional values of `cin` to valid state
  - Does not affect the `buffered input`
- `cin.ignore(numeric_limits<streamsize>::max(), '\n');`
  - Ignore contents in the buffer
    - First argument is the max number of characters to ignore
    - Second argument is a character that, when observed in the stream, tells us to stop ignoring characters

Notes

Overview

Notes

References

- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.

Notes