Stream Input/Output: A Deeper Look

Chapter 13

Introduction

C++ uses *type-safe Input/Ouput* (*I/O*).

Each *I/O* operation is executed in a manner sensitive to the data type:

- If an *I/O* function is defined to handle the particular data type, then that function is called
- If not, the compiler generates an error.

Hence, improper data cannot “sneak” through the system. It can occur in C and leads to some subtle and bizarre errors.

Introduction

C++ uses *type-safe Input/Ouput* (*I/O*).

Each *I/O* operation is executed in a manner sensitive to the data type:

- If an *I/O* function is defined to handle the particular data type, then that function is called
- If not, the compiler generates an error.

Hence, improper data cannot “sneak” through the system. It can occur in C and leads to some subtle and bizarre errors.

Recall that we had to overload the *output stream insertion operator* (<<) and *input stream extraction operator* (>>) for `class Complex`,`class Quadrilateral`, etc.

Streams

C++ (*I/O*) occurs in streams (sequences of bytes)

In input operations: the bytes flow from a device (a keyboard, a disk drive, a network connection) to main memory

In output operations: the bytes flow from main memory to a device (a display, a printer, a network connection)

These transfers usually take much more time that the time the processor requires to manipulate data internally.

Streams

C++ provides “low-level” and “high-level” I/O capabilities.

Low-level I/O (*unformatted I/O*): specify that some number of bytes should be transferred device-to-memory or vice-versa

High-level I/O (*formatted I/O*): bytes are grouped into meaningful units (integers, characters, string, floating-point numbers, user-defined types)

Programmers generally prefer the later, which are satisfactory for most I/O other than high-volume file processing.

Streams

C++ provides “low-level” and “high-level” I/O capabilities.

Low-level I/O (*unformatted I/O*): specify that some number of bytes should be transferred device-to-memory or vice-versa

High-level I/O (*formatted I/O*): bytes are grouped into meaningful units (integers, characters, string, floating-point numbers, user-defined types)

Programmers generally prefer the later, which are satisfactory for most I/O other than high-volume file processing.

Classic Streams vs. Standard Streams

Classic stream libraries supported only `char`-based *I/O*.

It limits the set of characters that can be displayed (one byte): see the ASCII character set.

Unicode is an extensive international character set that represents the majority of the world’s languages, math. symbols and much more.

Original C++ type for processing Unicode: `wchar_t`
Starting from C++ 11: `char14_t` and `char32_t`

Standard stream libraries are implemented as class templates and can be specialized for various character types

`<iostream>` header

`<iostream>` header declares the basic services required for all *stream I/O* operations.

Class templates:
`basic_istream` :   for stream input operations
`basic_ostream` :   for stream output operations
`basic_iostream`   provides both stream input and stream output operations

`<iostream>` header

`<iostream>` header declares the basic services required for all *stream I/O* operations.

Class templates:
`basic_istream` : for stream input operations
`basic_ostream` : for stream output operations
`basic_iostream` provides both stream input and stream output operations

We used:
`istream`, alias for `basic_istream<char>` , enables char input, and
`ostream`, alias for `basic_ostream<char>` that enables char output.
`iostream` is an alias for `basic_iostream<char>` that enables both char input and output

`<iostream>` header

`cin` has a type `istream` and is said to be predefined to be connected to the *standard input device* (which is often a keyboard)

```
int a;
cin >> a; // data “flows” in the direction of arrows
```

The compiler selects the appropriate overloaded *stream extraction operator* >>, based on the type of a.

`<iostream>` header

`cout` has a type ostream and is said to be predefined to be connected to the *standard output device* (which is often a display screen)

```
int a = 23;
cout << a; // data “flows” in the direction of arrows
```

The compiler selects the appropriate overloaded *stream insertion operator* <<, based on the type of a.

`<iostream>` header

`cerr` and `clog` are also a predefined object, an ostream object, and is said to be connected to the *standard error device*, normally the screen.

Outputs to object `cerr` are unbuffered, meaning that they appear immediately. *unbuffered standard error stream*

Outputs to object `clog` are buffered. It means that each insertion to `clog` could cause its output to be held in a buffer, until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique.

Stream Output

`ostream` provides *formatted* and *unformatted* capabilities

stream insertion operator `<<` allows output of standard data types

`put` member function provides output of characters, one-at-a-time

`write` member function provides unformatted output

and many others

See outputExamples.cpp

Stream Input

`istream` provides *formatted* and *unformatted* capabilities as well

stream extraction operator `>>` usually skips white-space character

After each input, the *stream extraction operator* returns a reference to the stream object that received the extraction message, however it can be used in a condition (with implicit conversion to `bool` type, starting from C++ 11).

get and getline

`get` member function extracts characters from the stream as unformatted input.

(1) single character

```
   int get();
   istream& get (char& c);
```

(2) c-string

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

(3) stream buffer

```
   istream& get (streambuf& sb);
   istream& get (streambuf& sb, char delim);
```

See details at
http://www.cplusplus.com/reference/istream/istream/get/

get and getline

`get` member function *with no arguments* inputs one character from the designated stream, including white-space character and other nongraphic characters and returns it as the value of the function call.

It returns `EOF` when the end of file is encountered in the stream.

`EOF` usually has a value of `-1` and is defined in a header that is indirectly included via stream library headers like `<iostream>`

get and getline

`eof()` is a member function of `cin`, checks for end-of-file. It returns `true` if the end-of-file is reached, and `false` otherwise.

Recall: <Ctrl> z (on Windows) and <Ctrl> d (on Linux and Mac)

get and getline

Let’s look at these examples:

- Using `cin`, `eof`, and `get` with no arguments: inputAndOutputExample.cpp
- Using `cin` and `get` with two arguments: inputAndOutputExample2.cpp
- Using `cin` and `get` with two and three arguments: inputAndOutputExample3.cpp

get and getline

Member function geline operates similarly to the
```
istream& get (char* s, streamsize n, char delim);
```
version of the get, but it removes the delimeter from the stream.

see inputAndOutputExample4.cpp

get and getline: final comments

- When working with `get` and `getline` and *c-strings*, the *null character* '\0' is inserted at the end of the input.

- `ignore` function receives two arguments: a designated number of characters to skip (1 by default) and the delimiter at which to stop ignoring the characters (`EOF` by default)

- `putback` function places the previously obtained character by `get` function back into the stream.

- `peek` function returns the next character from the input stream, but does not remove it from the stream

Unformatted I/O with `read`, `write` and `gcount`

- `read` (member function of istream) reads input bytes into a built-in array of chars; if fewer than the designated number of characters are read, `failbit` is set.

- `write` (member function of ostream) outputs bytes from a built-in array of chars

these bytes are not formatted in any way, they are input/output as raw bytes

- `gcount` (member function of istream) reports how many bytes were read by the last input operation

  see inputAndOutputExample5.cpp

Stream Manipulators

- We saw the following output stream manipulators so far:
  - `setw()`
  - `left`
  - `right`
  - `setprecision()`
  - `fixed`

Stream Manipulators

- `setprecision(10)` is used together with `fixed` if we would like to use the fixed-point notation of the decimal, rounded off to 10 decimal places

It is a *sticky* manipulator, so once set, it will continue to “work” until the setting is changed. No default parameter value.

Alternative: member function `precision()` of `ostream`.

```
double r{sqrt(7)};
cout << fixed << setprecision(10);
cout << r << endl;
cout.precision(8);
cout << r << endl;
```

A call `precision()` with no argument returns the current precision.

Stream Manipulators

- `setw(10)` stream manipulator sets the *field width*
- `width` member function (of classes `istream` and `ostream`) also sets the *field width*
- the `width()` call with no argument returns current setting

It is not a *sticky* manipulator, so it applies only to next insertion/extraction operation.

If output value is narrower than the width, it is right alligned.
If output value is wider than the width, it will not be truncated.

When using with *input stream*, one less symbol is read.

See outputStreamManipulation.cpp

Stream Manipulators

Integers are usually interpreted as decimal values (base 10). To change the base in which integers are interpreted ion a steam, insert the

- `hex` manipulator to set to base 16 (hexadecimal)
- `oct` manipulator to set to base 8 (octal)
- `dec` manipulator to set to base 10 (decimal)

It is a *sticky* manipulator.

A stream's case can also be changed by calling `setbase(n)` parameterized stream manipulator, where n = 16, 10, or 8.

Stream Manipulators

Section 13.7 has a table of stream manipulators with examples. Read it.

To return an output stream's format to its default state use `flags` member function:

- without parameters it returns the current format settings as an `fmtflags` data type, which represents the format state.
- when called with parameter `fmtflags` sets the format state as specified by the argument

See flags.cpp

Stream Error States

Each stream object contains a set of state bits that represents a stream’s state – sticky format settings, error indicators, etc.

We can test it through bits of class `ios_base` – the base class of stream classes.

Bits for input stream:
`failbit` is true if the wrong type of data is input
`badbit` is true if the operation fails in unrecoverable manner
`eofbit` is true if the end-of-file is encountered
`goodbit` is true if none of the bits above are set to true

After an error occurs, we can no longer use stream until we reset its error state. Use `clear` member function to do it.

See errorStates.cpp

Tying an Output Stream and an Input Stream

During an interaction process, we alternate prompt statements with output. The prompt should appear before the input operation proceeds.

With buffered output, output appears only:
- when the buffer fills
- when the outputs are flushed explicitly by the program
- automatically at the end of the program

We can use function `tie` to synchronize the operation of `istream` and `ostream` to ensure that output appears before the subsequent input.

```
cin.tie(&cout); // to tie

inputStream.tie(0); // to untie
```

In-class work

Exercise 13.8 (Printing Pointer Values as Integers)
Write a program that prints pointer values, using casts to all the integer data types.
Which one prints strange values?
Which one cause errors?

Visit https://en.cppreference.com/w/cpp/language/types to see the integer types.

HW assignment

1) Exercises 13.6, 13.7
2) recall the `class Complex`:
re-define the stream extraction operator (input stream) to be able to get the input in the form 4 – 9i from the user. It should determine whether the data entered is valid, and if it is not, it should set `failbit` to indicate improper input.

Self-Study:
read sections 13.7 and 13.8

Optional (for self-development):
Sections 13.6.4