Files and streams

Raul P. Pelaez

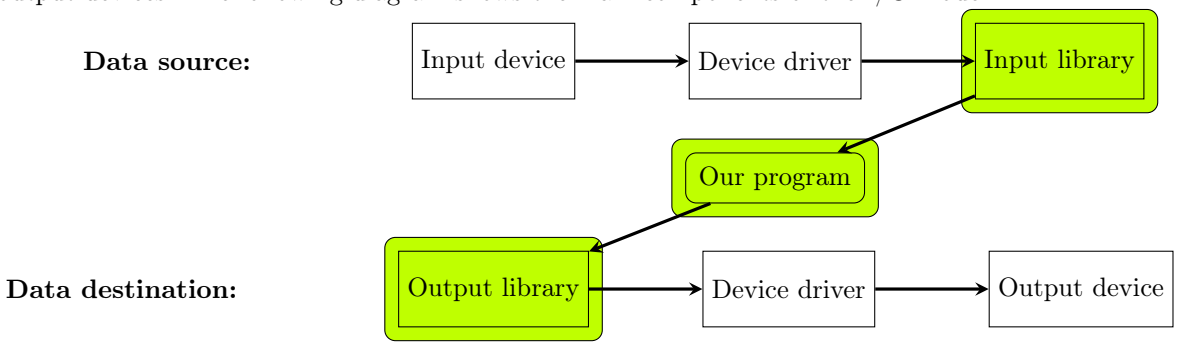
October 31, 2024

Contents

1 Input and output streams

In C++ input and output is done through streams. A stream is a sequence of bytes that can be read from or written to. Streams are a high level abstraction that allows to read and write data from different sources and to different destinations. The C++ standard library provides a set of classes that allow to work with streams. The most important classes are `std::istream` (input stream) and `std::ostream` (output stream) which are used to read and write data from and to streams respectively. These classes are used as base classes for other classes that provide more specific functionality.
You have already seen examples of streams via `std::cin` (an instance of `std::istream`) and `std::cout` (an instance of `std::ostream`). These are used to read and write data from and to the standard input and output (the terminal) respectively. The standard library also provides classes that allow to read and write data from and to files, under the same abstraction of streams.
In particular, `std::ifstream` and `std::ofstream` are classes that allow to read and write data from and to files respectively.

The C++ I/O model abstracts away much of the complexity of working with different input and output devices. The following diagram shows the main components of the I/O model:



Info

Everything is a file
One of the genius ideas of the UNIX system (Linux, OSX) is that everything is a file. Such as a file, a device, a network connection, a pipe, etc. This allows to use the same I/O model for all of them.
For instance, you can use `std::cin` to read from the terminal, but this is equivalent to opening the special file `/dev/stdin` and reading from it. Similarly, `std::cout` is equivalent to writing to `/dev/stdout`.

With a model like this we can write our program in a way that is independent of the actual device we are reading from or writing to. For instance, we do not have to concern ourselves about whether the input device is an HDD, and SAD, a network mounted drive, a recording device, etc. We just read from a stream and the system takes care of the rest.

1.1 Base stream

Info

`std::ostream`

- Turns values of various types into character sequences.
- Sends those characters "somewhere" (such as a console, a file, the main memory, or another computer).

Info

`std::istream`

- Gets characters from "somewhere" (such as a console, a file, the main memory, or another computer).
- Turns those characters into values of various types.

1.2 String stream

One example of an ostream/istream is the `std::stringstream` class. This class allows to read and write data to a string. This is useful when we want to convert data to a string or vice versa.
Example
The following code uses a `std::stringstream` to write data to a string and then print it to the terminal.

Source code

```cpp
#include <iostream>
#include <sstream>
#include <chrono>
int main() {
  int i = 42;
  double d = 3.14;
  auto now = std::chrono::system_clock::now();
  std::stringstream ss;
  ss << "The value of i is: " << i << std::endl;
  ss << "The value of d is: " << d << std::endl;
  ss << "The current time is: " << std::chrono::system_clock::to_time_t(now) <<
  ↪  std::endl;
  std::string all = ss.str(); // Get the string
  std::cout << all; // Use the cout ostream to print the string to terminal
    return 0;
}
```

1.3 File stream

The `std::ifstream` and `std::ofstream` classes allow to read and write data from and to files respectively. These classes are derived from `std::istream` and `std::ostream` respectively. This means that they can be used in the same way as `std::cin` and `std::cout`, file streams, string streams and the terminal streams are used in the same way.

Example: Writing a file
The following code writes a file in the current folder with some data:

Source code

```cpp
#include <iostream>
#include <fstream>
int main() {
  std::ofstream file("example.txt");
  if (!file) {
    std::cerr << "Error opening file" << std::endl;
    return 1;
  }
  file << "Hello world!" << std::endl;
  file << 42 << std::endl;
  file << "Some more text" << std::endl;
  file << 3.14 << std::endl;
  return 0;
}
```

Example: Reading a file
The following code reads that same file and prints its contents to the terminal:

Source code

```cpp
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream file("example.txt");
  if (!file) {
    std::cerr << "Error opening file" << std::endl;
    return 1;
  }
  // Read one line
  std::string firstWord, secondWord;
  file >> firstWord >> secondWord; // Hello world!
  // Read one int
  int i;
  file >> i; // 42
  // Read the rest of the file
  std::string line;
  while (std::getline(file, line)) {
    std::cout << line << std::endl;
  }
  return 0;
}
```

Advice

The beauty of RAII makes it so that we do not have to close the file explicitly. The file will be closed automatically when the `file` object goes out of scope.

Advice

Continuously reading from a stream
Say we want to create a kind of prompt, where our program reads from the terminal forever, responding to user input each time they press enter. We can do this by reading from the terminal in a loop, until the user types a special command to exit. The following code does this:

Source code

```cpp
#include <iostream>
#include <string>
int main() {
  auto ist = std::cin; // Could also be std::ifstream, or
  ↪  std::istringstream
  for(MyType var; ist>>var;) {
    do_something(var);
  }
  return 0;
}
```

Keep reading to learn how to make an user-defined type understand the >> operator.

1.4 Error handling

All streams provide some methods to check their state after an operation, the most relevant ones being:

- `good()`: The operations succeeded.
- `fail()`: Something unexpected happened (e.g, we looked for a digit and found 'x')
- `bad()`: Something unexpected and serious happened (e.g., a disk read error)
- `eof()`: We reached the end of the file.

Additionally, both `std::istream` and `std::ostream` overload the `operator!` to check if the stream is in a bad state.
Example

Source code

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream file("example.txt");
  if (!file) {
    std::cerr << "Error opening file" << std::endl;
    return 1;
  }
  // Read one line
  std::string firstWord, secondWord;
  if(!(file >> firstWord >> secondWord)) // Hello world!
    std::cerr << "File does not start with two strings" << std::endl;
  return 0;
}
```

1.5 User-defined input/output operations

The `std::istream` and `std::ostream` classes provide a way to define how to read and write user-defined types. This is done by overloading the `operator<<` and `operator>>` respectively. This allows to use the same syntax as with built-in types. For instance, if we have a `MyType` class, we can define how to read and write it as follows:

Source code

```
#include <iostream>
class MyType {
public:
  int i;
  double d;
};
std::ostream& operator<<(std::ostream& os, const MyType& mt) {
  os << mt.i << " " << mt.d;
  return os;
}
std::istream& operator>>(std::istream& is, MyType& mt) {
  is >> mt.i >> mt.d;
  return is;
}
int main() {
  MyType mt;
  std::cin >> mt;
  std::cout << mt;
  return 0;
}
```

Advice

Making the stream operators friends

In the previous example, the `operator<<` and `operator>>` functions are defined as free functions. This means that they are not part of the `MyType` class. This is necessary because the left-hand side of the operator is the stream, not the `MyType` object. However, we want this function to access the private members of the `MyType` class. To do this, we can declare the `operator<<` and `operator>>` functions as friends of the `MyType` class and even define them inside its body.

Example

Source code

```
#include <iostream>
class MyType {
  int i;
  double d;
public:
  friend std::ostream& operator<<(std::ostream& os, const MyType& mt) {
    os << mt.i << " " << mt.d;
    return os;
  }
  friend std::istream& operator>>(std::istream& is, MyType& mt) {
    is >> mt.i >> mt.d;
    return is;
  }
};
int main() {
  MyType mt;
  std::cin >> mt;
  std::cout << mt;
  return 0;
}
```

1.6 Formatting

There exists many "manipulators" that change the state of a stream. For instance, if we want to print a number in hexadecimal format, we can use the `std::hex` manipulator. The following code prints the number 42 in hexadecimal format:

Source code

```cpp
#include <iostream>
int main() {
  std::cout << std::hex << 42 << std::endl;
  return 0;
}
```

Other manipulators are:

- `std::dec`: Decimal format
- `std::oct`: Octal format
- `std::setw(n)`: Set the width of the next output to n characters
- `std::setfill(c)`: Fill the output with character c

And many more.
Additionally, we have the `std:format` function that allows to format strings in a similar way to the `printf` function in C or f-strings in Python. The following code prints the number 42 in hexadecimal format:

Source code

```cpp
#include <iostream>
int main() {
  std::cout << std::format("The number is: {:x}", 42) << std::endl;
  return 0;
}
```

Note that `std::format` is only available in C++20.

1.7 An example

The Cycles game uses streams to send packets of data between the server and the clients. The following code shows how the server sends a packet to the client, this is the method that sends the game state to all clients:

Source code

```
auto sendGameState(std::vector<sf::Socket> &clientSockets) {
spdlog::debug("Server ({}): Sending game state to {} clients", frame,
              clientSockets.size());
if (clientSockets.size() == 0) {
  return std::vector<Id>();
}
sf::Packet packet;
packet << conf.gridWidth << conf.gridHeight;
const auto &grid = game->getGrid();
auto players = game->getPlayers();
packet << static_cast<sf::Uint32>(players.size());
for (const auto &[id, player] : players) {
  packet << player.position.x << player.position.y << player.color.r
         << player.color.g << player.color.b << player.name << id << frame;
}
for (auto &cell : grid) {
  packet << cell;
}
std::vector<Id> successful;
for (const auto &[id, clientSocket] : clientSockets) {
  if (clientSocket->send(packet) != sf::Socket::Done) {
    spdlog::debug("Server ({}): Failed to send game state to player {}",
                  frame, id);
  } else {
    successful.push_back(id);
    spdlog::debug("Server ({}): Game state sent to player {}", frame, id);
  }
}
return successful;
}
```

Note how despite the fact that the server is sending data to a network socket, it uses the same syntax as if it was writing to a file or the terminal.

1.8 Key new concepts

Info

I/O stream model

- The I/O stream model abstracts away the complexity of working with different input and output devices by providing a common interface based on "streams".

Info

`std::ostream` and `std::istream`

- Base classes for output and input streams.
- Derived classes provide more specific functionality, such as:
  - `std::ifstream` and `std::ofstream` for reading and writing files.
  - `std::stringstream` for reading and writing strings.
  - `std::cin` and `std::cout` for reading and writing to the terminal.

Info

User-defined types and I/O

- User-defined types can be read from and written to streams by overloading the `operator<<` and `operator>>` operators.

2 Exercises

Create a new copy of your submission template. We are going to add some headers to the include directory and some tests to the test directory.
You can define and declare your classes and functions in the header file, but I recommend you to define them in a separate source file under src for practice.

2.1 Basic stream handling

Goal

Write the definition for the following function using string streams.

Source code

```
// include/split.h
#pragma once
#include <string>
#include <vector>
namespace homework{
/**
 * @brief Returns a vector of whitespace-separated substrings from the
 ↪  argument string
 * @param str The string to split
 * @return A vector of whitespace-separated substrings
 */
std::vector<std::string> split(const std::string& str);
} // namespace homework
```

Use the following test case to check your implementation:

Source code

```
#include "gtest/gtest.h"
#include "split.h"
using namespace homework;
TEST(Split, MultipleWordsWithExtraSpaces){
  auto res = split("  hello   world lorem ipsum  ");
  ASSERT_EQ(res.size(),4);
  ASSERT_EQ(res[0],"hello");
  ASSERT_EQ(res[1],"world");
  ASSERT_EQ(res[2],"lorem");
  ASSERT_EQ(res[3],"ipsum");
}
```

2.2 Serialization

Goal

Serialization is the process of converting an object into a format that can be stored or transmitted. In this exercise, you will implement a class that can be serialized to a file. You could use this kind of operation to, for instance, save the state of a game to a file or to import/export the configuration of a program.
Complete the following implementation by adding the missing operators and defining the necessary functions

Source code

```cpp
// include/mytype.h
#pragma once
#include <iostream>
#include <fstream>
namespace homework {
/**
 * @brief A type with some members that can be serialized
 */
class MyType {
    int i;
    double d;

public:
    MyType(int i, double d) : i(i), d(d) {
    }
    /**
     * @brief Equality operator
     */
    auto operator==(const MyType& other) const {
        return i == other.i && d == other.d;
    }
};
/**
 * @brief Write the MyType object to a file
 * @param mt The MyType object to write
 * @param filename The name of the file to write to
 */
void writeToFile(const MyType& mt, const std::string& filename) {
    // Write the MyType object to a file with the given filename
}
/**
 * @brief Read the MyType object from a file
 * @param filename The name of the file to read from
 * @return The MyType object read from the file
 */
MyType readFromFile(const std::string& filename) {
    // Read the MyType object from a file with the given filename and
    ↪  return it
}
} // namespace homework
```

Goal

Use the following test case to check your implementation:

Source code

```
//tests/test_mytype.cpp
#include "gtest/gtest.h"
#include  "mytype.h"
#include <fstream>
using namespace homework;
TEST(MyType, WriteRead){
  MyType mt(42,3.14);
  writeToFile(mt,"mytype.txt");
  auto mt2 = readFromFile("mytype.txt");
  ASSERT_EQ(mt,mt2);
}
```

hint

- You can use the `std::ofstream` and `std::ifstream` classes to write and read the data respectively.
- You must overload the stream operators for MyType.