

# VE280 Programming and Elementary Data Structures

Paul Weng  
UM-SJTU Joint Institute

## I/O Streams



# Learning Objectives

- Understand I/O
- Understand streams
- Know how to read/write from standard input/output, files, and strings

# Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream

# Input/Output

## Streams

- A popular model for how input and output is done in computer systems is centered around the notion of a **stream**.
- A stream is just a sequence of data with functions to put data into one end, and take them out of the other.

```
cin >> a;
```

# Input/Output

## Streams

- Typical streams:

keyboard	→	program
display	←	program
file	→	program
file	←	program
string	→	program
string	←	program

- In C++, streams are **unidirectional**.
- Data is always passed through the stream in one direction.
- If you want to read and write data to the same file or device, you need two streams.

# Input/Output

## Streams

- In general, there are two kinds of stream data: **characters** and **binary data**.
- Characters are usually used for:
  - Communicating between your program and a keyboard or screen.
  - Reading and writing text files.
- In addition to text, files can contain arbitrary **binary** data.
  - It is usually much more **efficient** than character representation.
  - However, it is hard to understand and debug.
- We'll talk about **character streams** here.

# Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream

# Output Stream: cout

```
cout << "Hello, world!\n";
```

- Output to screen.
- The << is called the **insertion operator**, and is used to insert things into the output stream.
  - It knows how to **convert** all of the other standard data types to **characters** before inserting them into the stream.

```
int foo = 42;
```

```
cout << foo << endl;
```

- Can be cascaded

```
cout << foo << " " << bar << endl;
```



# Print with Fixed Field Width

```
cout << foo << setw(4) << bar << endl;
```

- Here the `setw()` manipulator sets the width of the **following** number to the specified number of positions and **right-aligns** the number within that field.
- It pads with spaces.

right align 

			7
--	--	--	---

left align 

7			
---	--	--	--

- If you want to use `setw()`, you should  
`#include <iomanip>`

# Alternate Output Streams

- You can also use the Linux I/O **redirection** facility to move the output end of the stream from screen to a file:

```
$ ./hello > foo
```

- This connects the output end of the `cout` stream to the file “foo”.
- There is another output stream object defined by the `iostream` library called `cerr`.
- This stream is identical in most respects to the `cout` stream; in particular, its default output is also the screen.
- By convention, programs use the `cerr` stream for **error messages**.

# Output: Buffering

- I/O in C++ is **buffered**.
- This means output inserted into an output stream is saved by the underlying operating system (in a region of memory called a **buffer**).



- The content in the buffer is written to the output only when specific actions are taken.

# Output: Buffering

- The buffer content is written to the output only when:
  - A newline is inserted into the stream, i.e., **endl** or **'\n'**
  - The buffer is explicitly flushed. E.g.,  

```
cout << "ok" << flush;
```
  - The buffer becomes full
  - The program decides to read from `cin`
  - The program exits
- Once the buffer content is written to the output, the buffer is **cleaned**
- If some content is not printed out, it may be still in the buffer
- In contrast, output sent to `cerr` is not buffered

# Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream

# Input Stream: cin

- `cin >> foo;`
  - Takes input from keyboard
  - `>>` is called the **extraction operator**, and is used to extract things from the input stream.
  - Knows how to convert the characters you type into values of simple types and strings.
- Question: what are values of the variables?

```
int foo;  
double bar;  
string baz;  
cin >> foo >> bar >> baz;
```

Assume inputs is:  
42 3.14 four score\n

Note: baz is just “four”!

How to get baz as “four score”?

# getline()

- If you need to read strings including **blanks** or **tabs**, use the `getline()` function:

```
cin >> foo >> bar;  
getline(cin, baz);
```

Assume inputs is:  
42 3.14 four score\n

- `getline()` reads all characters **up to but not including** the next newline and puts them into the **string variable**, and then **discards the newline**
- But `baz` is “ four score”; it keeps the leading space

# get()

- The `get ( )` function reads **a single** character, whitespace or newlines:

```
char ch;  
cin.get(ch); // Extracts a character  
//from cin stream and stores it in ch
```

- So, we can accomplish what we'd hoped to accomplish by:

```
cin >> foo >> bar;  
cin.get(ch);  
getline(cin, baz);
```

Assume inputs is:  
42 3.14 four score\n

- This makes `baz` “four score”.

The three methods have such different syntax. However, the three methods can be freely intermixed.



# Input: Buffering

- Like `cout`, `cin` is **buffered**.
- Characters typed (which are to be gathered by `cin`) are stored in a buffer **until the enter key** is pressed.
- The characters are then made available to the program as a group.
- This also allows for greater efficiency, and it lets you correct errors before your program sees them (i.e. you can go back and fix something you typed wrong).

# Alternate Input Streams

- You can use the Linux I/O **redirection** facility to move the input end of the stream from the keyboard to a file:

```
$ prog < foo
```

- When doing this, remember that the input will not appear on your screen since you did not enter it on the keyboard.
  - This makes funny-looking output, as the input is not echoed.

# Failed Input Streams

- The extraction operator will fail if inappropriate data is given to it.
- For example, if:

```
int foo;  
cin >> foo;
```

is presented with:

```
42abc\n
```

the attempted conversion will succeed, up to the point of the “a”, i.e., `foo = 42`

- The stream will be left with “abc\n” in it.

# Failed Input Streams


```
int foo;  
cin >> foo;
```

- However, if you present it with something that **does not** begin with a digit, like:

abc

then the stream will enter a **failed** state.

- You can test the state of a stream by using it where a bool is expected:
  - For example, `if (cin) {...}`     `while (cin) {...}`
  - It returns **true** if it is **good**, false otherwise.
- A failed input stream will resist all attempts to extract more data from it

 After running the code below, the user inputs: “VE 280 is the best course ever!” What are the contents of the variables?

```
int x;  
string s1, s2;  
cin >> s1 >> x >> s2;
```

- **A.** s1 contains “VE”.
- **B.** x contains 280.
- **C.** s2 contains “is the best course ever!”.
- **D.** s2 contains “is the best course ever”.

# Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream

# File Streams

- Why use files?
  - Files allow you to store data permanently!
  - Data output to a file lasts after the program ends
  - An input file can be used over and over. No typing of data again and again for testing
- File stream: I/O between file and program
- Linux has I/O redirection facility. Then, why use file streams?
  - E.g., when you need to write to two files

# Using File Streams

- `#include <fstream>`
- Declare an **input** file stream object  
`ifstream iFile;`
- Declare an **output** file stream object  
`ofstream oFile;`
- The file stream object must be connected to a file
  - Connecting a stream to a file is opening the file for the stream  
`iFile.open("myText.txt");`



# Using File Streams

- Use the **input** file stream: use the extraction operator >> and the `getline()` function

```
int bar;  
iFile >> bar;  
string baz;  
getline(iFile, baz);
```

- Use the **output** file stream: use the insertion operator <<  
`oFile << bar;`

# Closing a File

- After using a file, it should be closed

```
file_stream.close();
```

- This disconnects the stream from the file.
- Why closing a file?
  - Close files to reduce the chance of a file being **corrupted** if the program terminates abnormally.
  - It is important to close an output file if your program later needs to read input from that output file
- The system will automatically close files if you forget as long as your program ends normally
  - ... but **explicitly** closing the file is recommended!

# Input File Streams

## Example

- Consider the following:

```
#include <iostream>
#include <fstream>
using namespace std;
void main() {
    ifstream iFile;
    int bar;
    iFile.open("foo");
    iFile >> bar;
    cout << "The answer is " << bar << ".\n";
    iFile.close();
}
```

- This opens the file named `foo` for reading, and associates it with the input stream object `iFile`.
- Thereafter you can extract input from the file in the same way we did using `cin`.
- If the file named "`foo`" contains the characters "`42`", this program will output:  
The answer is 42.

# Failed File Streams

- The file stream enters the failed state if:
  - It cannot be opened.
  - You attempt to read past the end of the file.
- A stream's state may be checked by evaluating the stream object:

```
if (iFile) { ... }
```

- A stream in the failed state will return false.
- Example

```
iFile.open("a.txt");  
if (!iFile) {  
    cerr << "Cannot open a.txt\n";  
    return -1;  
}
```

# Example of Reading File

```
while(iFile) {  
    getline(iFile, line);  
    cout << line << endl;  
}
```

Why not good?

How to correct this?

- Normally, after getline reads an entire line, iFile points at the position of the “\n”
- If getline reads the last line, iFile points to the end of file
  - **Note**: iFile is still good! So, the program will issue another getline, which reads nothing
  - So, the program will print an empty line
  - This time, iFile passes the end of the file and loop terminates

# Example of Reading File: Correction

```
while(iFile) {  
    getline(iFile, line);  
    if(iFile) {  
        cout << line << endl;  
    }  
}
```

# Example of Reading File

- Another much simpler (and correct) way

```
while (getline (iFile, line)) {  
    cout << line << endl;  
}
```
- `istream& getline(istream& is, string& str);`
- Return value: a reference to its parameter `is` (with the value after issuing the current `getline`).
- Question: why it works?

# Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream



# String Stream

## Motivation

- Suppose that you use the `getline()` function to read an entire line from a file and the result is stored in a string.

```
string line;  
getline(iFile, line);
```
- Suppose that the line contains an int followed by a double. We want to read these two numbers from the string `line`.
- We can use input string stream!
  - It reads characters in a string and convert them into values of proper types

# String Stream

## Motivation

- Suppose we have a string of a book name and an int of its published year. We want to create a string whose first part is the book name and the second part is its published year.
  - Notice that we need to convert the int to a string!
- We can use output string stream!
  - It writes to a string
  - It knows how to convert standard data types into characters and insert them into the string

# String Stream

- There are two types of string stream: **input** string stream and **output** string stream.
- C++ defines string stream in the sstream library  
`#include <sstream>`
- Declare an input string stream object  
`istringstream iStream;`
- Declare an output string stream object  
`ostringstream oStream;`

# Input String Stream

- When we use input string stream, it is usually assigned a string it will read from.

```
iStream.str(a_string);
```

- We can use extraction operator `>>` on an input string stream to retrieve the data.

```
istringstream iStream;  
int foo;  
double bar;  
iStream.str(line);  
iStream >> foo >> bar;
```

If `line` is the string  
“42 3.14”, then

```
foo = 42;  
bar = 3.14;
```

# Output String Stream

- We can use output string stream to format a string.
  - For example, we might have a collection of numeric values but want their string representation.
- We use insertion operator `<<` to insert characters into an output string stream.
- We fetch the string value of the string stream using the member function `str(void)` of a string stream.

# Output String Stream

## Example

```
int foo = 512;  
int bar = 1024;  
string result;  
ostringstream oStream;  
oStream << foo << " " << bar;  
result = oStream.str();
```

result is a string "512 1024".

# References

- **C++ Primer (4<sup>th</sup> Edition)**, by *Stanley B. Lippman, Josée Lajoie, Barbara E. Moo*, Addison-Wesley Publishing (2005)
  - Chapter 8.4 **File Streams**
  - Chapter 8.5 **String Streams**
- Course notes, pages 42-47