Formatted Output and Output Streams

CS 1044

Output Streams

`#include <fstream>`

- New data type: `ofstream`
- Read this in your head as **“output file stream”**
- We can declare variables that represent the file stream
- Then, we connect that stream to a file (possibly that doesn’t exist yet) that determines where the output will go

Opening an Output File

- Opening an output file:

```
ofstream myfile("output.txt");
```

- Can also do it this way:

```
ofstream myfile;
myfile.open("output.txt");
```

Writing Values to a File

- Writing values to an `ofstream` works just like it did with `cout` – just replace `cout` with **the variable name**

```
int a = 50;
double b = 4.9;
string c = "hello";
myfile << a << b << c << endl;
```

Basic Output

- Recall that by default, no formatting of output is performed
- Spaces aren’t inserted between values, doubles are printed to who-knows-how-many decimal places
- How do we make it look nicer?

Use Case: Generating Reports

- Imagine that you’ve read a large amount of data from a file and processed it
- Your output might be a **report** that should contain **tables** of data in **neatly aligned** columns
- If values in the columns are **different lengths** (e.g., numbers with different digits), figuring out the spacing by hand would be **tedious**

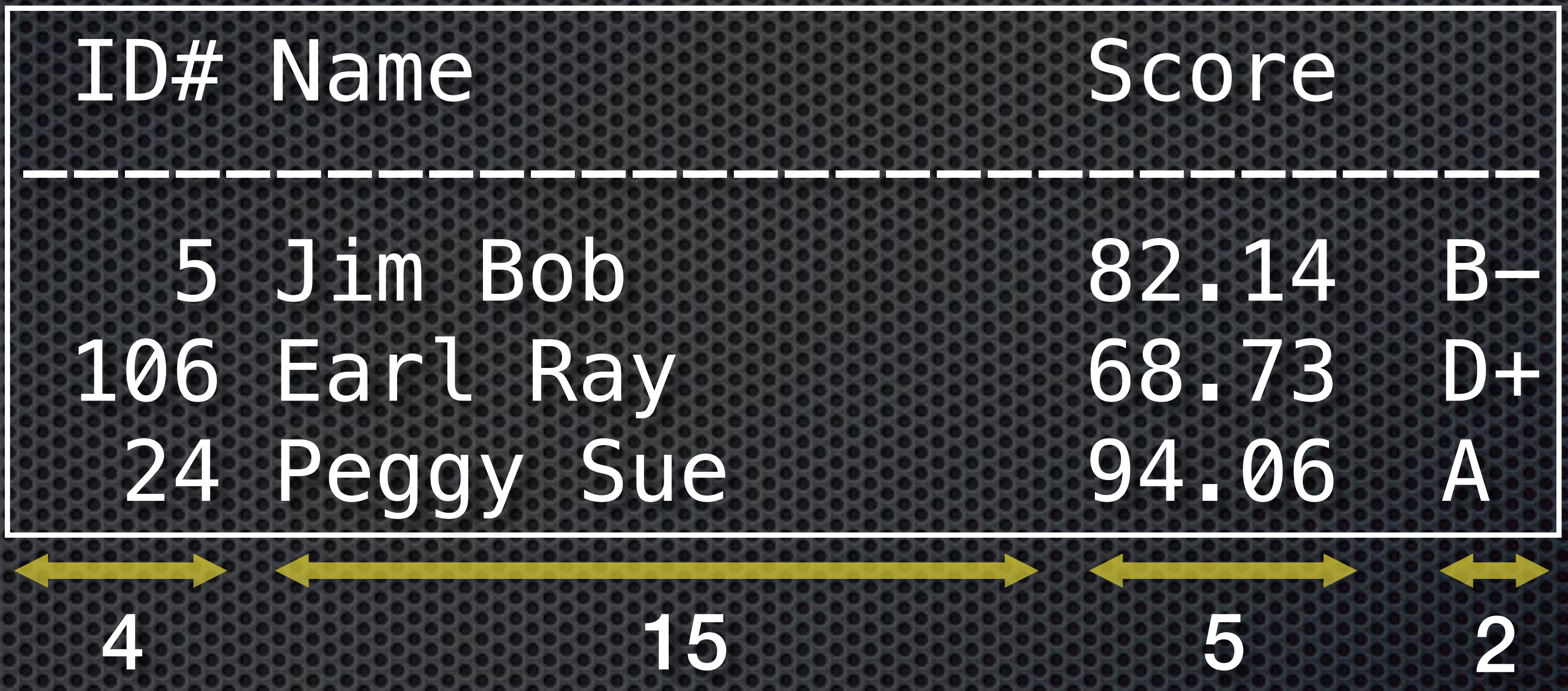
Output Manipulators

```
#include <iomanip>
```

- C++ provides **output manipulators** that you can insert into streams using <<
- Most manipulators don't generate output of their own, but **affect how future values are output**
- Some manipulators affect only the **one** next thing being output, others affect **everything** from there on out

Tabular Output

- Imagine that we wanted the following table:



```
 ID# Name            Score
------------------------------
   5 Jim Bob         82.14  B-
 106 Earl Ray        68.73  D+
  24 Peggy Sue       94.06  A
```

- We can talk about each field having a particular **"width"** in characters, and an **alignment** within that

Field Width

- We can output a value in a **field** that has a fixed width, **padded by spaces** if the value is smaller than the field

```
cout << setw(4) << id;
```

- By default, values are **right-aligned** in the field
- `setw` only applies to the **immediate next value** output
- Be careful: Values too wide will just **overflow** the field

Changing Alignment

- Use the `left` and `right` manipulators to control how a value is aligned in a field

```
cout << right << setw(4) << id;
cout << left << setw(15) << name;
```

- These are **"sticky"** – they affect **every** value output afterward, not just the next one
- If you're changing alignment a lot, it might be best to be explicit about the alignment of each field

Formatting Decimal Values

```
cout << fixed << setprecision(2) << score;
```

- Argument above to `setprecision` represents the number of **digits after the decimal point**
- `fixed` and `setprecision` are **“sticky”**
- If you leave out fixed, the argument determines the number of **significant digits** instead

Changing the Padding

- `setfill` changes the character used to fill the rest of a field if the value is too short

```
cout << setfill('.') << setw(15) << name;
```

- `setfill` is “**sticky**”, so pass a single space to `setfill` after you use it to turn it off