Basic Input/Output

CS 1044
Streams

- Basic input/output (I/O) in C++ is based on streams

- **Output stream** – an endless sequence of characters going to an output device

- **Input stream** – an infinite or finite sequence of characters coming from an input device
Built-in Streams

- C++ provides two streams we can use for programs with interactive I/O
  - `cout` – a stream that sends data to the console (the screen) by default
  - `cin` – a stream that reads data from the keyboard by default
Writing to a Stream

- To get information out of our programs, we might write it to a stream

- Use the insertion operator, `<<`

  ```
  cout << "x is equal to ";
  cout << x;
  ```

- Inserting a **string literal** will print that string exactly

- Inserting a **variable** will print the **value** of that variable
What Gets Printed?

- When you write a variable/literal to an output stream, the output depends on the type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code> or <code>string</code></td>
<td>The exact text value</td>
</tr>
<tr>
<td><code>int</code></td>
<td>The exact numeric value</td>
</tr>
<tr>
<td><code>double</code></td>
<td>By default, 6 significant digits; uses scientific notation if necessary</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>By default, 0 for <code>false</code> and 1 for <code>true</code></td>
</tr>
</tbody>
</table>
“Chaining” Stream Output

- It would be **annoying** if we had to put each thing we wanted to print on a separate line.
- C++ lets us **chain** the multiple `<<` operators together.

```cpp
cout << "x is equal to " << x;
```

- We can do this as many times as we want, as long as we start with an output stream on the left.
Basic Formatting

- C++ **will not** add any formatting or spacing to your output unless you explicitly tell it to

  ```cpp
  cout << "x is equal to" << x;
  ```

- Make sure you put spaces in your string literals where you need them

- Send `endl` to an output stream to move down to the next line
Reading from a Stream

- To get information into our programs, we might read it from a stream

- Use `cin` and the `extraction operator`, `>>`

  ```cpp
  cin >> x;
  ```

- Unlike output, the right-hand side of an input extraction should usually be a `variable` (with some exceptions)

  ```cpp
  cin >> "Hello"; // doesn't make sense
  ```
Interactive I/O

- If your program’s first action is to wait for input, it might appear to “hang” for the user.
- So before asking the user for input, you should usually output a meaningful prompt telling them what to do:

  ```
  cout << "Please enter your age: ";
  cin >> age;
  ```
“Chaining” Stream Input

- We can chain input using the extraction operator `<<` just like we can chain output.
- Both of the following examples would read a value into `a`, then a value into `b`, then a value into `c`:

```
cin >> a >> b >> c;  // First example

// Second example:
cin >> a;
cin >> b;
cin >> c;
```
What Gets Read?

- Like output, C++ is “smart” about reading input depending on the type of the variable it will be stored in.
- But input is a bit more complicated than output.
- Example: What if you try to read a value into an int variable but the next thing in the input stream is "Barney"?
If you write

```
cin >> x;
```

- First, any leading whitespace is skipped
- Then, characters are read as long as they make sense for the type of the variable `x` is
### Whitespace

- Characters that don’t produce a visible image on the screen are called whitespace.

<table>
<thead>
<tr>
<th>Description</th>
<th>As a char</th>
<th>In a string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single space</td>
<td>' '</td>
<td>&quot;hello world&quot;</td>
</tr>
<tr>
<td>Horizontal tab</td>
<td>'\t'</td>
<td>&quot;hello\ntworld&quot;</td>
</tr>
<tr>
<td>Line break</td>
<td>'\n'</td>
<td>&quot;hello\nworld&quot;</td>
</tr>
</tbody>
</table>
endl vs. '\n'

- `endl` and `\n` are not exactly the same thing.
- Think of `endl` as a "command" that can be sent to an output stream to move to the next line.
- `\n` is how C++ represents the `character` that `endl` generates.
- `endl` can only be used with streams; `\n` can be used in anywhere a `character` is needed.
endl vs. '\n'

- Even though `endl` and `\n` are not exactly the same, the three lines below do produce the same output:

```cpp
cout << "Hello world" << endl;
cout << "Hello world" << '\n';
cout << "Hello world\n";
```

Using `\n` as a separate character

Embedding `\n` in a larger string
What Gets Read?

- Once the whitespace is skipped, how does the variable’s type determine what is read from the stream?

<table>
<thead>
<tr>
<th>Type</th>
<th>Input Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>The next single character</td>
</tr>
<tr>
<td>string</td>
<td>The next sequence of characters until a space or the end of input is encountered</td>
</tr>
<tr>
<td>int</td>
<td>The next integer (stopping when something that isn’t 0–9 is encountered; do not enter commas!)</td>
</tr>
<tr>
<td>double</td>
<td>The next double (stopping when something that is encountered that wouldn’t be a valid decimal number)</td>
</tr>
<tr>
<td>bool</td>
<td>By default, if the next value is a non-zero integer, it reads true; otherwise, it reads false</td>
</tr>
</tbody>
</table>
Input Failure

- If we try to read something incompatible into a variable, we get **input failure**. Consequences:
  - The program **keeps running**, but the stream enters a “failed” state
  - Later >> operations execute but **do nothing**; the variables being read into **do not change**
  - **Cannot read** successfully again until the failure state is “cleared”
Error Checking

- In general, you would want to do error checking, gracefully handle input failure, try to recover.
- Sometimes this can be difficult, especially if you don’t have control over the input coming into your program.
- For assignments in this class, I won’t intentionally give you improperly formed input.
More Advanced Input

- Sometimes you need more control over input than what `>>` provides, especially when reading strings.
- Remember that `>>` stops when any whitespace is reached.
- What if you want to read an entire line, or read text until a different character is reached?
Reading Whole Lines

```cpp
string s;
generateLine(cin, s);
```

- Reads text from a steam (in this case, `cin`) until the
  next line break is reached
- Puts the result in the `string` variable given as the
  second argument
- Unlike `>>`, `getline` does not skip leading whitespace
Stopping Elsewhere

```cpp
string s;
getline(cin, s, ':');
```

- Optional third argument lets us specify a different **stopping character**
- The example above would read text until a colon is encountered (or the end of the stream)
- Stopping character is **not included** as part of the result, and the next input operation will begin **after** it
Ignoring Some Input

`cin.ignore(count, '\n');`

- Skips up to `count` (an integer) characters or until a line break is encountered, *whichever comes first*
- Can use any character as stopping character, like `getline`
- It’s common to see this

  `cin.ignore(INT_MAX, '\n');`

if you want to skip whole lines and aren’t sure how long they are – *2 billion* is a pretty safe bet