Topic 2

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files

Reading Words and Characters

What really happens when reading a **`string`**?

```
string word;
in_file >> word;
```

1. Any whitespace is skipped
   (whitespace is: `'\t' '\n' ' '`).
2. The first character that is not white space is added to the string **`word`**.
   More characters are added until either another white space character occurs, or the end of the file has been reached.

You can read a single character, including whitespace, using `get()`:

```
char ch;
in_file.get(ch);
```

The **`get`** method returns the "not failed" condition so:

```
while (in_file.get(ch)) //reads entire file, char by char
{
   // Process the character ch
}
```

Lookahead: Reading a Number Only If It Is a Number

You can look at a character after reading it and then put it back. This is called *one-character lookahead*. A typical usage: check for numbers before reading them so that a failed read won't happen:

```
char ch;
int n=0; //for reading an entire int
in_file.get(ch);

if (isdigit(ch)) // Is this a number?
{
   // Put the digit back so that it will
   // be part of the number we read
   in_file.unget();

   data >> n; // Read integer starting with ch

}
```

Functions in <cctype> (Handy for Lookahead): Table 1

Function	Accepted Characters
isdigit	0 ... 9
isalpha	a ... z, A ... Z
islower	a ... z
isupper	A ... Z
isalnum	a ... z, A ... Z, 0 ... 9
isspace	White space (space, tab, newline, and the rarely used carriage return, form feed, and vertical tab)

Reading A Whole Line: `getline`

The function

`getline()`

reads a whole line up to the next '\n', into a C++ string. The '\n' is then deleted, and NOT saved into the string.

```
string line;
ifstream in_file("myfile.txt");

getline(in_file, line);
```

NOTE: to read a line into a C-string, the syntax is different.

`getline` becomes a member function of the stream object:

```
char cline[100];
in_file.getline(cline, 99); //the int argument
// is the max number of chars to read
```

Reading A Whole Line in a Loop: `getline`

The **`getline`** function, like the others we've seen, returns the "not failed" condition.

To process a whole file line by line:

```
string line;
while( getline(in_file, line)) //reads whole file
{
   // Process line
}
```

Processing a File Line by Line

Here is a CIA file from
`http://www.cia.gov/library/publications/the-world-factbook/`

Each line has: country name, its population, a newline character.

(And there is some whitespace in there too.)

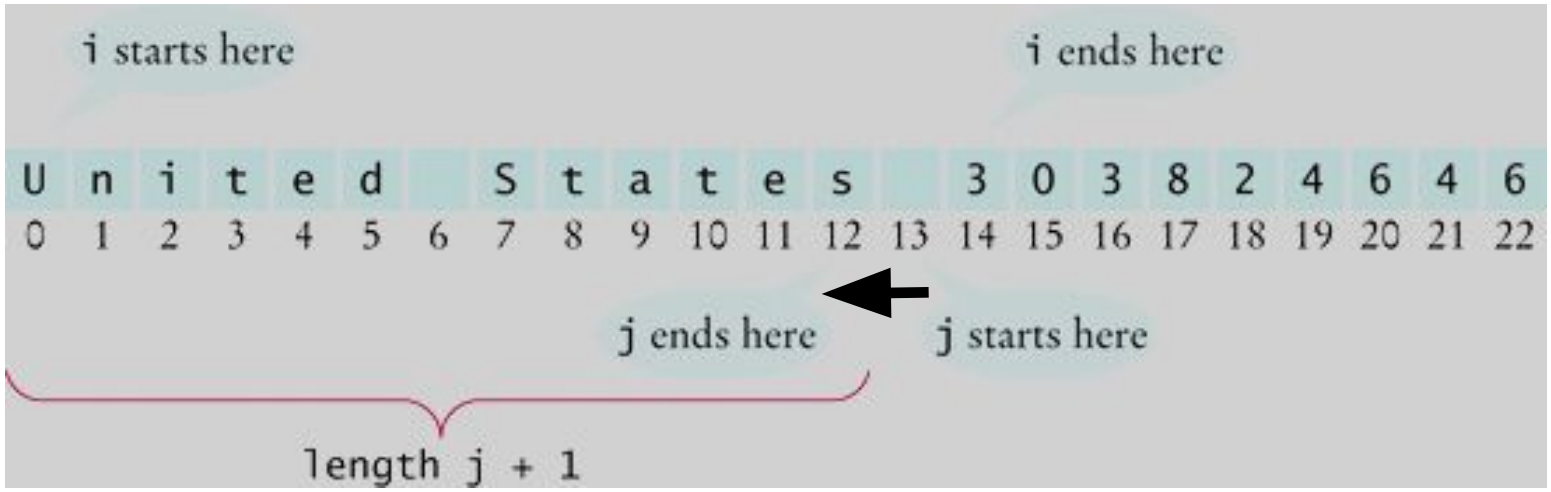
```
China 1330044605
India 1147995898
United States 303824646
...
```

We'll read the file line by line with `getline`.
To extract the data from a **`line`** string, we need to find out where the name ends and the number starts.

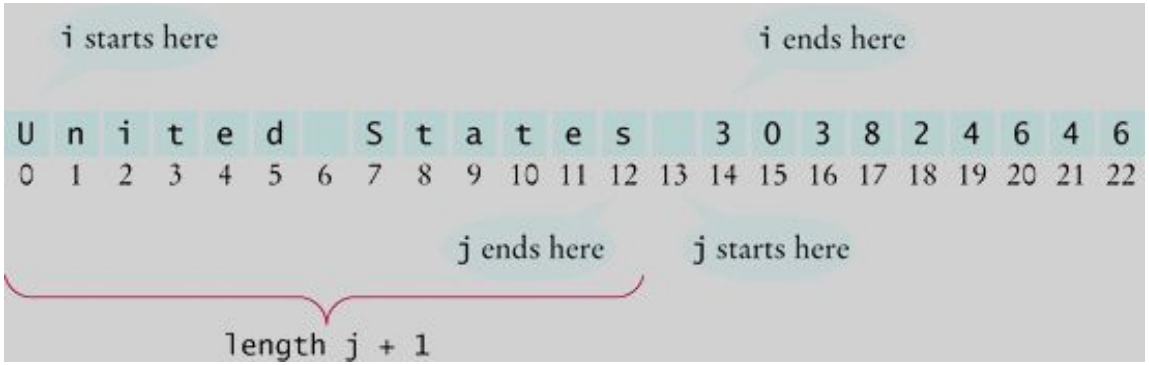
Parsing a File Line using <cctype> Functions

```
// Locate the first digit of population
int i = 0;
while (!isdigit(line[i])) { i++; }

// Go backwards to find end of country name
int j = i - 1;
while (isspace(line[j])) { j--; }
```



Capturing the Line characters into separate strings



Finally, extract the country name and population:

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```

There is just one problem: population is stored in a string, not a number.

You will see in Section 8.4 how to extract the population number as an `int`

Common Error: Mixing >> and `getline` Input

A problem occurs when a call to `getline` follows a `>>`

- >> does not remove any trailing `'\n'` from the input stream buffer, though it does skip a leading `'\n'` until it finds non-whitespace
- The `getline` reads only the newline left by the preceding >>, considering it as the end of an empty line.

- This would happen if the following code were used on a file that had lines of strings interleaved with lines of numbers:

```
while(!in.fail())
{
   getline(in, country_name); //gets empty strings
   in >> population;
}
```

The Fix for Mixing >> and `getline` Input

To solve the problem, any trailing newline in the input buffer must be removed before calling `getline`

We do this by adding a dummy `getline` after the `>>` statement:

```
string dummy;
while(!in.fail())
{
   getline(in, country_name); //gets empty strings
   in >> population;
   getline(in, dummy); //delete the dangling '\n'
}
```

There are other alternative remedies for deleting the dangling newline, including calling `in.get()` or `in.ignore()`, but we will leave those as research exercises for the reader.

Practice It: Line and Character Input

```
ifstream in;
string str;
char ch;
```

Write statements to carry out the following tasks (answers shown in tiny font):

Set str to the next line in the input.	`getline(in, str);`	The getline function reads the next line from the stream and places it in the string str. The newline is not part of the result.
Set str to the next word in the input.	`in >> str;`	The >> operator reads the next word from the stream, removing any whitespace that precedes it.
Set ch to the next character in the input, skipping any whitespace.	`in >> ch;`	Use the >> to get the next character from the stream, skipping any whitespace.
Set ch to the next character in the input, but do not skip whitespace.	`in.get(ch);`	Use the get function to get the next character from the stream, even if it is whitespace.
Fill in the condition to check whether ch is a digit: `if (...) { num = ch - '0'; }`	`isdigit(ch)`	The cctype header has functions for testing whether a character is a digit, letter, or whitespace.

Topic 3

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files

Writing Text Output

You use the operator **>>** to send **strings** and numbers to an output stream, and the **put** function for a single char:

```
string name = "Hello";
int value = 2;
char ch = '!';

ofstream out_file;
out_file.open("output.txt");
if (out_file.fail())
   { return -1; }

out_file << name << " " << value << endl;

out_file.put(ch);
```

Formatting Output – Manipulators

To control how the output is *formatted*,
you use *stream manipulators*.

A *manipulator* is an object that:

- is sent to a stream using the >> operator.
- affects the behavior of the stream.

Manipulators to Display a Time such as 09:01

Use **`setfill`** when you need to pad numbers with leading zeroes.

To set the width in which to display, use **`setw:`**

```
strm << setfill('0') << setw(2) << hours << ":"
<< setw(2) << minutes << setfill(' ');
```

That last **`setfill(' ')`** re-sets the fill back to the default space character.

Stream Manipulators: Table 2

Manipulator	Purpose	Example	Output

`setw`	Sets the field width of the next item only.	`out_file << setw(6) << 123 << endl << 123 << endl << setw(6) << 12345678;`	`  123`
			`123`
			`12345678`

`setfill`	Sets the fill character for padding a field. (default is a space.)	`out_file << setfill('0') << setw(6) << 123;`	`000123`
`left`	Selects left alignment.	`out_file << left << setw(6) << 123;`	`123`
`right`	Selects right alignment (default).	`out_file << right << setw(6) << 123;`	`  123`
`fixed`	Selects fixed format for floating-point numbers.	`double x = 123.4567;` `out_file << x << endl << fixed << x;`	`123.457` `123.456700`
`setprecision`	Sets the number of significant digits for the default floating-point format, the number of digits after the decimal point for fixed format.	`double x = 123.4567;` `out_file << fixed << x << endl << setprecision(2) << x;`	`123.456700` `123.46`

Practice It: Formatting Output

Produce this output to the `ofstream strm` (answers shown in tiny font):

```
12.345679 123456789.000000
```

```
strm << setprecision(6) << fixed << 12.3456789 << " " << 123456789.0;
```

(column width is 10):

```
       123
      4567
```

```
strm << setw(10) << 123 << endl << setw(10) << 4567;
```

```
----------|----------
Count:              177
----------|----------
```

```
strm << "----------|----------\n"
   << left << setw(10) << "Count:"
   << right << setw(11) << 177
   << "----------|----------\n";
```

Floating Point Formats: `fixed, scientific, defaultfloat`

For money values, choose fixed format with two digits after the decimal point.

```
out_file << fixed << setprecision(2);
```

To get the scientific format: a number with one digit before the decimal point and an exponent. As with the `fixed` format, the `setprecision` denotes the number of digits after the decimal point. :

```
out_file << scientific << setprecision(3) << 123.456;
```

produces

```
1.235e+02
```

To switch back to the default floating-point format, use the `defaultfloat` manipulator introduced in C++ 11:

```
out_file << defaultfloat;
```

Unicode, UTF-8, and C++ Strings

- The Unicode standard encodes alphabets from many languages, and some icons.
- Each Unicode character is 21-bits, written in hexadecimal for humans viewing the code
- For example,
  - é (Latin small letter e with acute accent) has the code U+OOE9
  - And  (high speed train) has the code U+1F684.
- For efficiency, characters in files or transmitted over the Internet are saved at less than 21 bits.
  - **each Unicode character saved as a sequence of one or more bytes.**
- In your programs, you can use the \U prefix followed by 8 hex digits for such characters:

```
string e_acute = u8"\U000000e9";
string high_speed_train = u8"\U0001f684";
```

- Use the `find` member function to look for a Unicode character:

```
string message = . . .;
size_t pos = message.find(e_acute);
if (pos != string::npos)
{
   // Message has e_acute starting at position pos
}
```

The `size_t` type indicates a non-negative position, and `string::npos` is a special value that denotes no position.