Getting Data from Files

CS 1044

Motivation

- Programs typically need to process **large amounts** of data
- It's not always appropriate to require the user to enter it all by hand in the console
- Data could be stored in files, come across a network connection, or from another external device...

File Streams

- Recall: `cin` and `cout` are **streams** that are attached to the keyboard and the screen (essentially)
- We can also attach streams to **arbitrary files** in order to read data from them and write data to them
- File streams can behave in subtly different ways to console streams

File Streams

```
#include <fstream>
```

- New data type: `ifstream`
- Read this in your head as **"input file stream"**
- We can declare variables that represent the file stream
- Then, we connect that stream to a file that contains the data that we want to read

Opening an Input File

- Opening an input file:

```
ifstream myfile("input.txt");
```

- Can also do it this way:

```
ifstream myfile;
myfile.open("input.txt");
```

Using `strings` as Filenames

- Watch out when you're using a `string` variable to store the filename:

Won't compile

```
string filename = "input.txt";
ifstream myfile(filename);
```

- Must apply a **special conversion** first:

```
string filename = "input.txt";
ifstream myfile(filename.c_str());
```

Closing a File

- Traditionally, we **close** a file when we're **done using it**:

```
myfile.close();
```

- We can do this explicitly, but C++ streams are **automatically closed** at the end of the variable's lifetime (typically at the end of the function it is declared in)

Reading Values a File

- Reading values from an `ifstream` works just like it did with `cin` – just replace `cin` with **the variable name**

```
int a;
double b;
string c;
myfile >> a >> b >> c;
```

- The same rules apply for reading values into variables of different types, how whitespace is ignored, etc.

Nature of the Input

- We've mostly worked with programs that dealt with a small amount of fixed input from the keyboard
- When working with files
  - What if we're processing a **large** amount of **repetitive** data?
  - What if we **don't know** ahead of time **how much data** is there to be processed?

Console Input vs. File Input

- If you try to read from `cin` but there isn't any more data, the program **waits for the user** to type something in (**interactivity**)
- Reading from files isn't interactive, since all the data is already there
- If you try to read from a file **after reaching the end**, then **input failure** results

Testing for Input Failure

- We can use just the name of a stream variable as the condition of an `if` statement or `while` loop

```
if (myfile) ...
```

- Will evaluate to `true` if the stream is still valid, or to `false` if input failure previously occurred

Input Example

- Simple example: a file containing a list of integers
- We want to compute the sum of all of the numbers in the file

```
100
94
87
93
90
68
75
100
82
```

First (Incorrect) Attempt

- Input failure doesn't occur until **after** you try to read something that you can't, like past the end of file
- If `myfile` is empty, it still evaluates to true at first
- So the loop would still run once when it shouldn't

```
int n;
int sum = 0;

while (myfile)
{
    myfile >> n;
    sum += n;
}
```

Fixed: Priming Read

- We can fix the code in the last slide by making an initial read **before** going into the loop
- Called a "**priming read**"
- Ensures that the loop **does not** execute an extra time if input failure occurs

```
int n;
int sum = 0;

myfile >> n;

while (myfile)
{
    sum += n;

    myfile >> n;
}
```

Try to read the first number

Will be false if nothing is left

Try to read the next number

General Outline

- Structure your input loop like this and you'll be **golden**:

```
read some data;

while (the stream is valid)
{
    process the last data read;
    read some data;
}
```

Notice that these are the same

Miscellaneous Input Topics

- `ifstreams` support other functions that you saw earlier with `cin`:

`getline(myfile, s)`	Reads everything up to the end of the line from `myfile` into the string variable `s`.
`getline(myfile, s, 'x')`	Reads everything up until `'x'` is reached from `myfile` into the string variable `s`.
`myfile.ignore(n, 'x')`	Ignores the next n characters in `myfile`, or until `'x'` is reached, whichever comes first.

Miscellaneous Input Topics

```
#include <iomanip>
```

- Sometimes you want to **explicitly skip** over all of the whitespace from the current read position
- For example, `getline` **doesn't automatically skip** leading whitespace, so you'd want to do this first:

```
myfile >> ws;
```

- The `ws` manipulator "eats" all upcoming whitespace, stopping when another character is reached

Passing Streams to Functions

- If you pass a stream into a function, you **must pass it by reference**

```
void foo(ifstream input);     // wrong
void foo(ifstream& input);    // right
```

- Passing a stream by value will make your program fail in **mysterious** ways when you try to read from it

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;
```