Lecture 19: Streams, Intro to File I/O

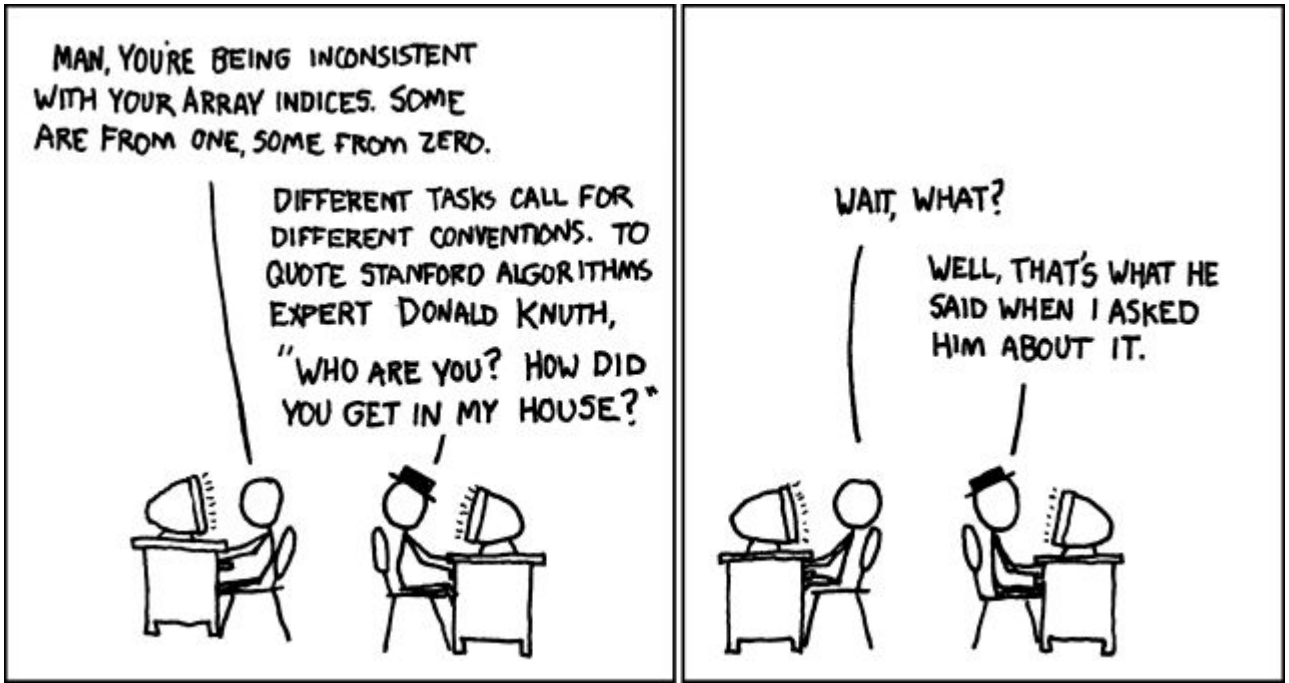
Announcements and reminders

Submissions:

- HW 6: Monday 6 PM
- Interview grading!
  By 3 March.

Last time on *Intro Computing…*

- We saw how to store groups of similar data using **arrays!**
  - … how to declare and define **2D arrays**!
  - … how to pass arrays as arguments into functions!



Streams

Aaahhh… a delicious ***stream*** of sushi…

(on conveyor belt)

- One at a time
- ***Input*** to your belly

Streams

Aaahhh… a delicious ***stream*** of sushi…
(on conveyor belt)

- One at a time
- ***Input*** to your belly
- Eventually, no more sush
(restaurant closes)

Reading and writing files

The C++ input/output library is based on the concept of ***streams***

- An **input stream** is a source of data
- An **output stream** is a destination of data

- The most common sources and destinations for data are the files on your hard disk
    - Need to know how to read/write disk files to work with large amounts of data that are common in business, administration, graphics, audio, science/math programs

Here’s a stream!

This is a stream of characters. It could be from the keyboard or from a file. Each of these is just a character - even these: 3 -23.73 which, when input, can be converted to: ints or doubles or whatever type you like.

(that was a '\n' at the end of the last line) &*@&^#!%#$ (No, that was –not- a curse!!!!!!!!!! ¥1,0000,0000 (price of a cup of coffee in Tokyo) Notice that all of this text is very plain - No bold or green of italics – just characters – and whitespace (TABs, NEWLINES and, of course... the other one you can’t see: the space character: (another '\n')

(&& another) (more whitespace) and FINALLY: Aren't you x-STREAM-ly glad this is over?

Here's a stream! … of characters!

```
This is a stream of characters. It could be from the keyboard  or  from a file.
Each of these is just a character - even these:  3  -23.73 which, when input, can
be converted to: ints or doubles or whatever type you like.

(that was a '\n' at the end of the last line) &*@&^#!%#$ (No, that was –not- a
curse!!!!!!!!!! ¥1,0000,0000 (price of a cup of coffee in Tokyo) Notice that all
of this text is very plain - No bold or green of italics – just characters – and
whitespace (TABs, NEWLINES and, of course... the other one you can't see: the
space character: (another '\n')

(&& another) (more whitespace) and FINALLY: Aren't you x-STREAM-ly glad this is
over?
```

- This stream is just a plain text file
  → No formatting, no colors, no video or music or other sound effects
- Can read these sorts of **plain text streams** of characters from the keyboard, as we have done so far using **cin >>**

Reading and writing files

You can also read and write files stored on your hard disk (drive)

- Plain text files
- Binary information (a binary file)
    - Images, audio recordings

To read/write files, you use ***variables*** of the **stream type**:

- **ifstream** for input from plain text files (input file stream)
- **ofstream** for output to plain text files (output file stream)
- **fstream** for input an output
- `#include <fstream>`

Opening a stream

To read anything from a file stream, you first must **open the stream** (same goes for writing)

- Opening a stream means associating your stream variable with the disk file
- First step is having a stream variable ready: declare it!

```
#include <fstream>
...
ifstream in_file;
...
// do stuff with the variable in_file...
```

Looks suspiciously like every other variable definition we have had…

… and it is! It's just a new variable type, to hold information we need to read from a file (or write to a file for `ofstream`)

Opening a stream

```
#include <fstream>
ifstream in_file;
in_file.open("input.txt");      // filename is input.txt
```

Just like with other variables, we can **declare** and **initialize** in one fell swoop:

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

When your program runs and tries to find this file (`input.txt`) it **will only look in the current directory that you're running the program!**

→ Common source of errors.

→ If desired file is not in the executing program's folder, can specify full file path

File names

If the file name comes from the user, you will store it in a **string**

If you use a C-string (**char[] array**), the open() function will also work fine

If you use a C++ string, some older library versions require you to convert it to a C-string using the <filename>.c_str() member function as the argument to open():

```
cout << "Please enter the file name: ";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str());
```

File names

If the file name is in a **char[] array**…

```
cout << "Please enter the file name: ";
char filename[80];          // instead of string, have 80 chars to store file name
cin >> filename;
ifstream in_file;
in_file.open(filename);
```

Closing a stream

When the program ends, all streams that have been opened will be automatically closed

It's good practice to manually close your streams with the **<filename>.close()** member function:

```
in_file.close();      // no argument needed
```

Reading from a stream -- you know more than you think!

Turns out, you actually already know how to read and write using files!

When we did
```
string name;
double value;

cin >> name >> value;
```

we got 2 inputs from cin (keyboard) and put them into the variables name and value

Let's get variables from an **input file stream instead!**

```
string name; double value;
ifstream in_file;
in_file.open(filename);
in_file >> name >> value;
```

No difference when it comes to reading using >>

Reading from a stream -- you know more than you think!

The >> operator returns a “not failed” condition

→ Allows you to combine an input statement ***and a handy-dandy test!***

- If the read failed, then the >> statement yields a **false**
- If the read was successful, then you get a **true**

```
if (in_file >> name >> value) {
    // process input in some way...
}
```

Reading from a stream -- you know more than you think!

We can even read ***all*** of the data from a file

→ Running out of things to read will cause the “failed state” in our little test!

- If the read failed, then the >> statement yields a **false**
- If the read was successful, then you get a **true**

```
while (in_file >> name >> value) {
    // process input in some way...
}
```

Failing to open

The open() method also sets a “not failed” condition.

It’s a good idea to test for failure immediately:

```
in_file.open(filename);
// check for failure after opening:
if (in_file.fail()) {
    return 0;
}
```

What just happened?!

We just saw…

- … what streams are!
- … how to open a stream!
  - and how to check if open() worked right!
- … how to close a stream!
- … how to read some data from a stream!

Next time:

… we read lots more data from streams!