MTH 4300, Lecture 11

Input Filestreams;
Stringstreams;
Object-Oriented Programming;
First Class Example: Mario;
Disappointing Mario Game

E Fink

March 10, 2025

1. Input Filestreams

In your homework, you saw the basics of output filestreams. There are also *input* filestreams. Here are the basics:

- You `#include <fstream>` to use.
- You declare an input filestream variable by, e.g.
  `std::ifstream file_var("actual_file_name.txt");`
  where `actual_file_name.txt` is the name of a file which must exist in your working directory! (You can use a relative or absolute path name to put your file in a different directory.) If no such file exists, then `file_var.fail()` will evaluate to `true`.
- One adds characters to the filestream's buffer just like with `cin`, except that characters enter the stream from the contents of the file rather than from the user's keyboard. When there are no more characters, your filestream is put in "end of file" state.
- Finally, once all code involving the file has been executed, you should close your file, by
  `file_var.close();`
  which will ensure that your operating system no longer thinks that the file in questions is still open.

L11x1_read.cpp

Input Filestreams

Frequently, one wants to have a program read the entire contents of a file line-by-line, stopping only once the end of the file has been reached. An idiom for this is:

```
while(std::getline(file_var, x)) {...
```

(where `x` is again a `string` variable). Essentially, `std::getline()`, in addition to writing a line from the file into `x`, returns a `bool` value which is `true` until the end of file is reached, when it returns `false` (and doesn't write anything into `x`). So, when you hit the end of the file, the loop stops automatically.

You can also replace the `getline()` call with any other read operation you wish to repeatedly execute until end-of-file (e.g. `file_var >> x`, which would read one WORD at a time); the loop will execute the operation until there is input failure, then the loop will stop.

L11x2_line.cpp

2. Stringstreams

Stringstreams are another type of stream. They don't involve keyboard, file, or screen input/output: instead, they are useful for converting data from one format to another. You use them as both input and output streams: you input values which are stored as a stream of characters, and then you can use stream extraction to output parts of that stream to other variables.

- You `#include <sstream>` to use.
- You declare a stringstream variable by, e.g.
  `std::stringstream my_stream;`
- Then, use stream operators and methods, like >>, <<, `getline()`.
- If you wish to recycle a stringstream variable for later use, it is adviseable to "reset and empty," using
  `my_stream.clear();`
  `my_stream.str("");`
  The latter fills `my_stream` with the empty string.

L11x3_sstream.cpp

Stringstreams

Let's use stringstreams to print out the sum of the numbers on each line of a file named `numbers.txt`. We'll grab each line from the file, using `getline()` within a `while` loop.
For each line, we'll pass it into a stringstream.
Then, we'll loop through that stringstream, extracting "words" (really, integers) from the string one at a time.

L11x4_sumline.cpp

3. Object-Oriented Programming and Classes

Object-oriented programming was developed to make big projects more manageable.

What *is* object-oriented programming? Well, there's a lot to it, but at a first pass, it's about creating your own data types that are meant to mimic how we think about real world objects.

The phrase "data abstraction" should be kept in mind: this refers to separating the interface of a data type from the implementation.

Once we have carefully built our data types, it will make writing programs which involve the relevant data much easier.

Object-Oriented Programming and Classes

You are already acquainted with the benefits of data abstraction. If you want to add 123 and 456, do you write code that converts these numbers to binary, then adds them bit-by-bit, then converts the binary back to base-10? No!

But all of that stuff is happening behind the scenes – someone *implemented* that functionality. You now have the privilege of not having to do that hard work: you can just call upon the simple *interface* – i.e, type `123 + 456`, the way you are accustomed to – and get the results you want. That way, you can focus on the harder problems you really care about.

Typically, when you build a class, you're trying to give yourself that same privilege. The main differences are:

- it's likely that the data you'll be interested in are a little but more multi-faceted than plain integers; and
- before enjoying the freedom of a simple interface, you'll probably have to put work in to designing the implementation. (But it will be worth it once you have that interface!)

Object-Oriented Programming and Classes

Classes are "user-defined" data types (where "user" refers to a user of the programming language, not a user of the end program).

The instances of these data types will be called *objects*. By "instance", we typically mean variables, although we can create objects without names too (for example, using pointers).

(e.g. `string` is a class, and in
`string x = "Hello";`
both `x` and `"Hello"` are objects)

Each object will be a big piece of data, which typically comprises a number of smaller bits of data. These will be called *attributes*, *data members*, or *member variables*. You should think of these attributes as representing the current **state** of each object.

In our programs, we will initialize, read data from, and change our objects via special functions called *methods* or *member functions*. You should think of these functions as representing the **behaviors** of objects: the things that they do, or are done to/with them.

4. First Class Example: Mario

We'd like to introduce the process of building and using classes. Logically, it makes sense to first build a class, and then use it. **However**, it is easier to appreciate the construction of a class if one has worked with instances of a class first.

Therefore: **L11x5_mario.cpp** . Focus on `main()`, but maybe quickly glance at the rest.

Context: For the sake of definiteness, I'm using the original Super Mario Brothers as the inspiration for my characters.

- Characters start off with 3 lives, and 0 coins
- When they get 100 coins, the coin count goes back to 0, but the number of lives goes up by 1
- When lives go down to 0, it's game over

(We'll ignore any graphical components, map location, abilities – our Mario game is going to be pretty unimpressive.)

First Class Example: Mario

What have we learned from this example?

- Declarations of class objects typically take the form
  `Classname var_name;`
  or
  `Classname var_name(initialization_value);`
  Every time you create an object, some code is run (a *constructor*) which initializes that object's member variables.
- If an object has the name `var_name`, then you can access member variables by `var_name.attr_name` (at least if the member variables are public – we'll discuss what this later).
- And you call methods related to that object by `var_name.attr_name()`.

5. Our Own Disappointing Mario Game

Ok, now we're going to make a silly game! In this game:

- the player enters their name.
- at each turn, a random number is selected: if it is 1,2,3, the player loses a life; if it is 4 or 5, the player collects 20 coins; if it is a 6, the player collects 50 coins.
- after each turn, player info should be printed out.
- when the game is over, the number of turns is printed out.

The use of methods should make this a little easier to write, and a lot easier to read! The intent of the code should be clear from the code itself.

L11x6_game.cpp