MTH 4300, Lecture 10

Dynamic Memory; Lifetime and Deallocation; Some Hazards of Pointers; Jagged Arrays

E Fink

March 4, 2025

1. Jagged Arrays

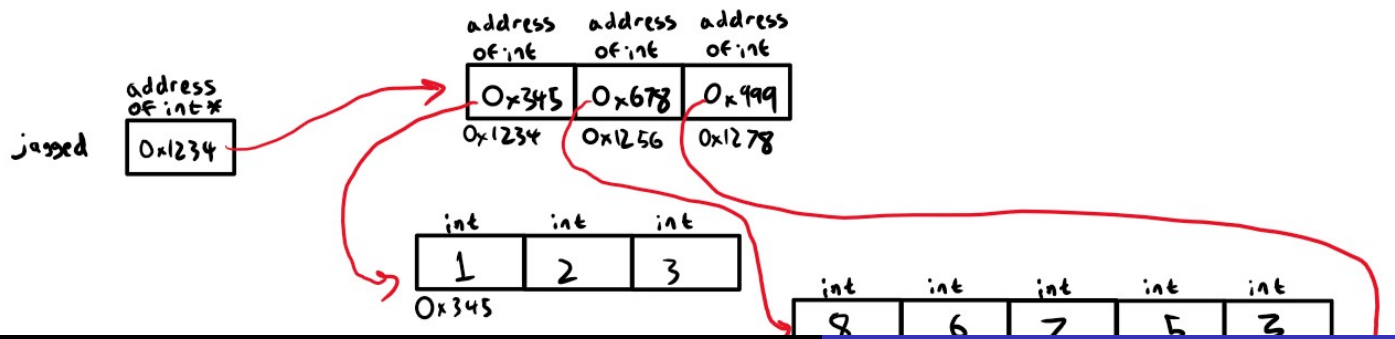
We've seen 2D arrays before, which have a fixed "height" and "width". In particular, the arrays contained within the array has the same length. But what if you want a *jagged array*: an array containing differently-sized arrays?

One can use a **double** pointer: e.g.

```
int **jagged;
```

To interpret this, `jagged` will be a pointer, but instead of being a pointer to an `int`, it is a pointer to an `int*`. Therefore:

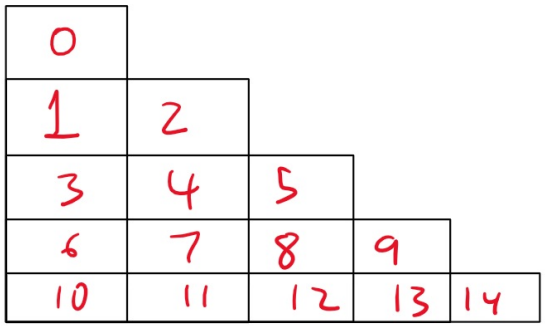
- `jagged` can hold the address of (the first entry of) an array of `int*`s
- Each entry holds the address of (the first entry of) an array of `int`s



Jagged Arrays

L10x2_tri.cpp

Create an array `triangle` with 5 entries, each of which is an array of `int`s. The first entry is an array of length 1, the second entry is an array of length 2, and so on. I recommend a SINGLE for-loop for this.



Then, fill the array with entries like the picture shows. For this one, I recommend a NESTED for-loop.

Oh, and we should probably deallocate the array as well!

2. Streams, `cin`, and >>

A *stream* is an object which can hold a sequence of characters, as they move from some sort of input to some sort of output. These include `cout`, `cin`, input and output filestreams, and stringstreams. Let's focus on `cin` for a moment.

At all times, `cin` maintains a *stream buffer* of characters. This can be thought of as a list of characters which starts out as empty at the beginning of a running program. Periodically, characters will get added to the end of this sequence (when new characters are input), and characters will also get removed from the front of this sequence (when the input values are appropriately written to variables).

Other stream variables share this general behavior: characters get added to the end, and removed from the front. What differs for other streams is where entering characters come from, and where exiting characters go to.

Streams, `cin`, and >>

If `x` is a variable of built-in type (e.g. `int`, `double`, `char`, etc.), how does the line `std::cin >> x;` behave?

- If `cin` is in a fail state (see below), the line is basically ignored.
- Otherwise, any *leading* whitespace in the stream – spaces, tabs, newlines, and other similar characters – will get removed from the stream and discarded. After that, the first character in the stream should not be whitespace.
- If the first character does not make sense for the datatype of `x` (e.g., if `x` is an `int` and the first character in the stream is a letter), then `cin` enters a *fail state*, and all subsequent calls involving it will not work; `x` will often be set to a special value, like `0`.
- Otherwise, `x` will get filled by removing characters from the stream: as many characters as make sense for the data type of `x`, or until a whitespace character is encountered, whichever comes first.
- What if the `cin` stream is empty? Then, the program pauses, and waits for the user to type in some input and press Enter.

Streams, cin, and >>

L10x3_cin.cpp

What will happen if we enter:

- `  1 2.0 a hello`
- `12.0hello`
- `x 1 2.0 a hello`
- `1`
  `2.0`
  `a`
  `hello`

Don't forget about `std::getline(cin, x)`, where `x` is a `string` variable. The function will extract every character, spaces and all, from the `cin` stream until the newline character is seen, and place them all into `x`. (The newline character won't be put into `x`, but will be removed from the stream.)

3. Rescuing cin

Since `cin` is prone to failure, it is useful to be able to restore it to working order. This can be delicate. The following member functions, which also work with other input streams, can help:

- First, `cin.fail()` returns a boolean value, which can be used to detect whether `cin` is in a fail state.
- `cin.clear()`: the `cin` stream contains several "status bits," which contain info about whether the stream is in good working order. The `.clear()` method essentially resets those bits to "all good."
- `cin.ignore(1000,'\n')`: after resetting state, the old, bad characters that were not successfully extracted will remain in the stream. However, you can flush them out using the command above, which will throw out 1000 characters **or** every character until a newline is encountered, whichever comes first. This makes a good follow-up to `cin.clear()`.

  Actually, an alternative which doesn't rely on a random long length is
  `cin.ignore(INT_MAX, '\n');`
  where `INT_MAX` is in the `<limits>` header.

L10x4_rescue.cpp

4. 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.

L10x5_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.

L10x6_line.cpp

5. 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.

L10x7_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.

L10x8_sumline.cpp