stream — sequence of data elements made available over time.

↳. provide a way to read & write data in a continuous flow, rather than in discrete chucks of data.

- TYPES: input stream & out put stream
  - input: use for reading data
    - ↳ classes includes ifstream & istringstream
  - output: use for writing data
    - ↳ classes includes ofstream & ostringstream
- buffered v.s. unbuffered:
  - buffered: data is collected in a buffer before reading & writing. can help improve performance by reducing the amount of I/O operations
  - unffered: data is read & written immediately. can be useful for real time applications.
- sequential access: order
  - force data to be read sequencially in a linear & orderlly manner (one piece at a time)
    - ifstream
  - typically access the data in the order it was written/stored.
    - ex:
      - processing large datasets that does not need the whole set to load & data order is important.
        - ↳ reading logs & processing files
      - NOTE: setting up a stream to access data a specific order will not allow user to jump to arbitrary position with a stream unless specified ahead

Questions ???

- does this means continuous intake of data? does this also implies that any data we code is process 1 by 1?
- continuous flow v.s. discrete chucks what are some examples of each?

- continuous intake of data / "available over time"
  - ↳ the way data is processed in a sequence (progressively) rather than all at once.
  - laymen's example:
    - when you're watching a video online or live, you are watching it one piece of the video at a time as the video loads. We didn't need to wait for the full length of the video to load.
  - examples: networking streams
    - when receiving data over a network, the package arrive continuously, & you process each packets as they arrive rather than waiting for the whole data set.
  - NOTE: sequentially & continuously does not imply that data will be process 1 by 1 all of the time.
  - for non-continuous flows, you would process the data in batches like: arrays, list, or maps, etc

- abstration: simplified
  - making a data set uniform or more straightforward.
  - unified interfaced (sanitization)
    - istringstream
  - hiding complexity
  - flexibility (switching type of data sources)
    - replacing a file input stream w/ a string input stream w/ minimal adjust.
  - polymorphism
    - allow functions to accept streams w/o needing to know their specific type.

Questions ???

- what are some sample cases where hiding complexity is useful?
- different examples of polymorphism vs flexibility.

istringstream allow a **string** input to be treated like a **stream**

↳ - to initialize **istringstream**

```
std:: istringstream name (input1 input2 input3)
```

whatever name we want.

could be a string input from getline or a direct "string"

* each space inbetween each input will be consider 1 data

- for data extract uses **example** →

```
int main() {
    std::istringstream iss("42 3.14 Hello"); // Create the istringstream object
    int x;
    double y;
    std::string s;

    // Use iss to extract data
    iss >> x >> y >> s; // x gets 42, y gets 3.14, s gets "Hello"

    std::cout << "x: " << x << ", y: " << y << ", s: " << s << std::endl;
    return 0;
}
```

if you know the data type you're expecting, be sure to declare them

```
data types name1;
data types name2;
data types name3;
```

```
name >> name1 >> name2 >> name3;
```

(for input1) (for input2) (for input3)

orders matter!

- for state management use **example** →

```
int main() {
    std::istringstream iss("42 3.14");
    int x;
    double y;

    // Attempt to read
    if (iss >> x >> y) {
        std::cout << "Read successfully: " << x << ", " << y << std::endl;
    } else {
        std::cout << "Error reading data." << std::endl;
    }

    // Check the state
    if (iss.eof()) {
        std::cout << "Reached end of input." << std::endl;
    }

    return 0;
}
```

↳ • tracking the state of the status of the **stream**

• **if statement** is involve

• func. to use for checking

good ( ) : (goodbit) stream is good state for I/O operations.

fail ( ) : (failbit) input operation failed to read data

bad ( ) : (badbit) serious error occurred. (memory issues)

eof ( ) : (eofbit) the end of stream has reached

- for resetting the **stream**.

↳. allow to clear the error states & reuse the **stream** with new data

• **clear ( )** func. to clear stream state's flag

str ( ) func. to assign new **strings**

example →

```
int main() {
    std::istringstream iss("42 3.14");
    int x;

    // Read data
    iss >> x;
    std::cout << "First read: " << x << std::endl;

    // Reset the stream
    iss.clear(); // Clear the state flags
    iss.str("100 200"); // Set new input data

    // Read again
    iss >> x;
    std::cout << "Second read: " << x << std::endl;

    return 0;
}
```

`while ( ) { };` while the condition remains true, execute code.

↳ `if statement` w/ if statement, if the condition is true, the code will move on.

- w/ `while loop`, the code will enter a loop & execute the same code.
- NOTE when coding `while loop` remember to code an exit. otherwise, we will be stuck in a ∞ loop.
- application
  ↳ force user to do something before they can move on.

`do { } while ( );` if the `do { }` condition is true, then enter the `while loop`.

- application
  ↳ play again option at the end of the game
    - to submit your hw, you must have proper file name.

`for ( ) { };` excecute the code x amount of times.

↳ in `( )` we can have up to 3 statements

1st: counter index or `i` (`index = #` or `i = #`)
this will tell the code where to start counting.

2nd: stopper counter `i =, >, <, >=, =<` #
count from `i = #` until `i =, >, <, >=, =<` #

3rd: increments `i++` (by +1) `i--` (by -1)
`i+= #` or `i-= #`

- `break` & `continue` statements

```
if ( i == # {
   break;
}
```

if the condition is true break from loop

if condition is true, skip pass it & continue with the loop