Structure

Brief Summary

The program is made of a few fundamental elements.

Mode detection

Detects and set the working mode (interactive or batch).

- **Interactive** - Reads input from `stdin` and prints to `stdout` printing a prompt before the user can enter new input.
- **Batch** - Command line arguments will specify the name of the input file and the name for the output file.

Batch Mode

! In batch mode, `stdout` and `stdin` are **redirected** so any function printing to `cout` will actually write to the output file.

The main loop

aka `main(argc, argv)` and `calcRunner(istream, ostream)`.

Each iteration prints a prompt (on interactive) + reads a line from `stdin` or the given input file.

Then, it applies the rest of the elements on it:

1. **`parseTree()`** - parses into a parse tree (`BinTree`).
2. **`executeTree()`** - Applies the parsed operations on the actual objects.

Parser

Input Sanitizer

aka `getNextToken(input = "", peak = false, expect_filename = false, tok_type_ptr)`.

Takes the input line (string) and converts it into tokens.

The first call initializes the function. Each consecutive call will then return the next token in the input line.

For the input `"g2 = g1 + {x,y,z}"` the tokens will be `["g2", "=", "g1", "{x,y,z}"]`.

Special options:

- **`peak`** - Will peak the next token (returning it), without advancing the position to the next token in line.
- **`expect_filename`** - Whether a filename token is expected (useful because a single filename could contain characters that would usually be split into different tokens).
- **`tok_type_ptr`** (Super optional) - Can be passed when the function initializes (non empty `input` is passed). Before a token returns, will set the variable it points to to the token type

(i.e. filename, graph literal, etc...).

Parse Tree Building

Iterates over the tokens and builds the co-responding syntax tree.

Also detects syntax errors.

Syntax Tree Runner

aka `executeTree(tree)`.

Will iterate recursively over the syntax tree, converting string values into actual graph objects, either from the `StorageManager` (graphs saved to a predefined variable), a `load` command or a graph literal (e.g. `{x,y,z|<x,y>}`).

Classes and Modules

`Exceptions` (Module containing multiple classes)

See all the Exceptions section for detailed explanation.

`Parser` (Module)

Handles parsing an input line into a binary syntax tree. Each node is a string token. A leaf would be an operand (graph literal, graph variable name or filename), other nodes will always be operators.

Syntax is validated in this stage.

Commands like `who`, `reset` and `quit` will be returned as a tree with one node, containing the command string.

Empty lines are considered `null` value expressions. Therefore, can only be a top level command (can't be an operand).

The `quit` function is handled outside the `executeTree` due to it's uniqueness.

`Graph` (Class)

The library which handles graph's functionality.

Can be used as a standalone library.

`GraphCalc` (Class)

Implements the functionality of running consecutive commands in the same environment.

Contains storage for graphs.

`StorageManager` (Class)

Manages storage for graph variables.

Implements functionality that affects program-wide memory.

It's member functions are:

- `get` - Retrieve a variable's value.
- `set` - Set a variable's value.
- `reset` - Reset the storage.
- `who` - Print to `cout` the list of variables (see Batch Mode).
- `remove` - Remove a single graph from storage.

`BinTree` (Class)

Used to save the input in a structured tree to easily execute it later on.

Other Classes and modules

There are a few more modules but they're less interesting.

Read the code it you want.

Exceptions

The `GraphCalc` library defines a few exceptions for its own use (and for users using `Graph` as a C++ library).

All exceptions derive from `GraphCalcError`.

The constructors of all the exceptions **require** an error message (`std::string`) to throw with the exception. So it is usually detailed.

Here are exceptions:

- **`GraphCalcError`** - The most general exception. Error message (as returned from `what()`) always starts with `"Error: "`.
- **`SyntaxError`** - When the syntax of the command is incorrect (e.g. unbalanced braces or something like `g = ++`).
- **`InvalidFormat`** - When something is defined with correct syntax but is still an invalid definition (e.g. self loops in a graph).
- **`MultipleDeclarations`** - More specific than `InvalidFormat`. When you define the same thing twice (e.g. When you define the node `x` twice in the same graph).
- **`Missing`** - When a definition of something is missing (e.g. when you try printing a graph that was not defined).
- **`Unknown`** - Anything that doesn't fit any of the other exceptions.

Author Notes

The project is programmed in a pretty modular way. Most elements can be taken out an reused in some other project.

That include the whole logic of the calculator. The calculator can be used without the friendly user interface (by simply instantiating a `GraphCalc` and passing it functions one by one).

You can also use the `executeTree` directly, passing it valid syntax trees. Note that in this case, you will have to pass it an instance of `StorageManager` that you create.

I would like to end with a quote:

> "When I get sad, I stop being sad and be awesome instead."
> -- Barney Stinson