Project 2: vtdraw

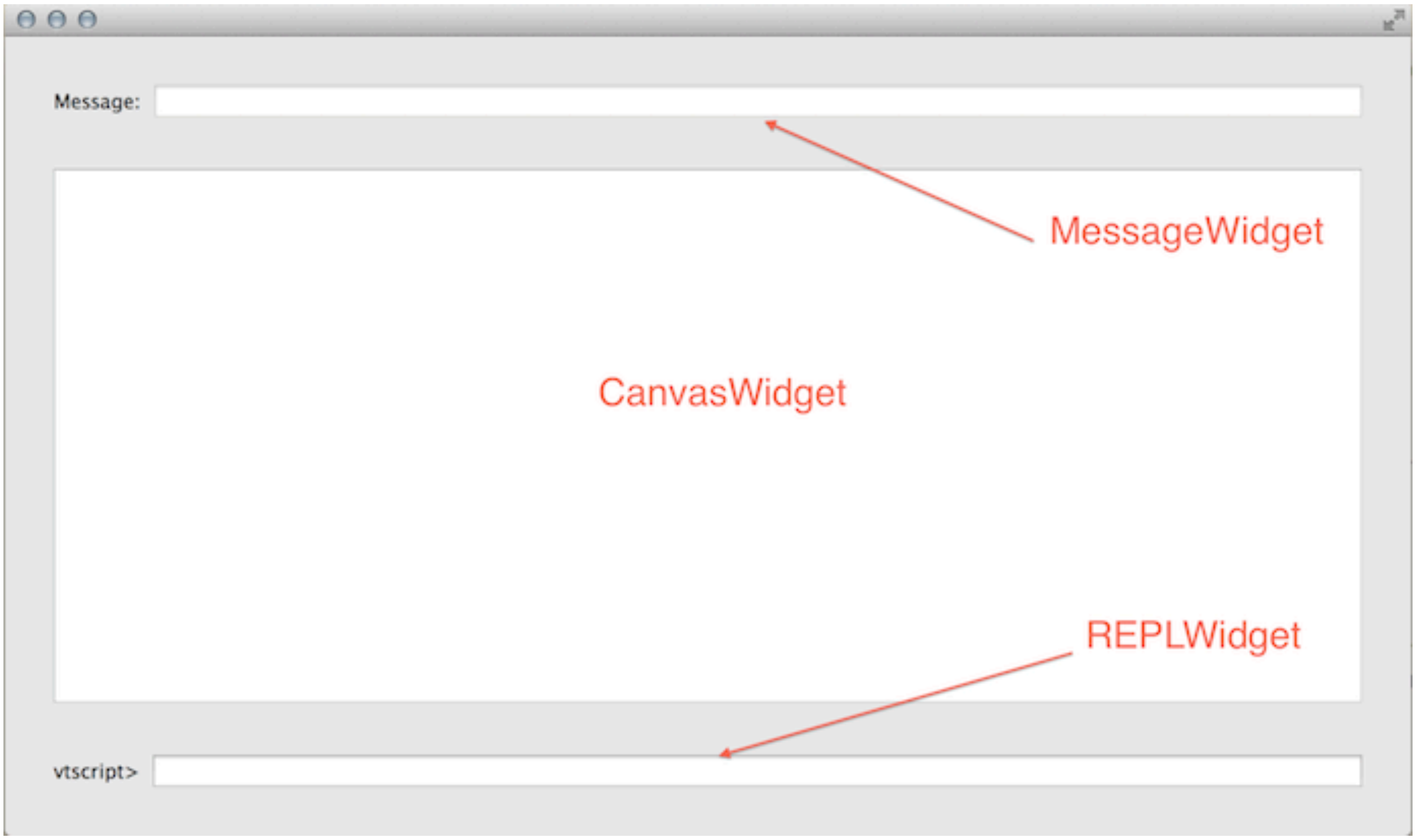
Assigned:	3/14
Beta Version Due:	3/28 at 8 am
Final Version Due:	4/11 at 8 am

GitHub Invitation Link (https://classroom.github.com/assignment-invitations/c5ebba163d63f867dee9bbf5884d17a4). Accept the GitHub invitation, wait until you get an email saying the import is complete, and then clone the git repository to your local machine.

The goal of this project is to gain practice designing and implementing a Qt application using a variety of design techniques including dynamic polymorphism, composition, and event-driven programming. You will be extending your interpreter from project 1 and embedding in into a simple drawing application we will call vtdraw. You can reuse any of your code from project 1.

Introduction

Recall from project 1 that languages with simple syntax make good scripting languages, interpreted languages that allow a user to interact with a larger application written in a compiled language like C++. Our application is a simple drawing viewer that looks like the following (with annotations):



It consists of a single window with 3 widgets embedded inside it arranged vertically. The top-most widget is a read-only message output window, where informational and error messages can be displayed. Below it is a canvas widget that displays the current drawing. At the bottom is a line-based input widget that acts as a read-eval-print-loop interface.

Language Specification for the extended vtscript

We will be extending our vtscript language from project 1 to include graphical types and procedures that have visual side effects. These extensions are:

- an Expression type Point with a value of two coordinates of type Number. Points are rendered as filled circles of radius two when drawn.
- an Expression type Line with a value of two Points.
- an Expression type Arc with a value of two Points and a Number. The first Point is the center of the arc, the second is the starting point of the arc, and the Number is the spanning angle in radians.
- an m-ary special form `draw` taking m arguments of a graphical type: Point, Line, or Arc and returning an Expression of type None. This form draws each argument when in a graphical environment, upon successful completion of a call to `eval()`. It has no visual effect otherwise.
- a built-in binary procedure `point` taking two Numbers and returning a Point
- a built-in binary procedure `line` taking two Points and returning a Line
- a built-in tertiary procedure `arc` taking two Points and a spanning angle in radians, and returning an Arc

In addition, the default environment also defines these procedures

- `sin`, unary expression of Numbers, returns the sin of the argument
- `cos`, unary expression of Numbers, returns the cos of the argument
- `arctan`, binary expression of Numbers, returns the arctan of the arguments, where the first is the y coordinate and the second is the x coordinate

A clarification/addition from project 1 is that Numbers should be compared for equality using the absolute difference compared to the machine epsilon to account for floating point rounding. See std::numeric_limits::epsilon (http://en.cppreference.com/w/cpp/types/numeric_limits/epsilon).

See the directory `tests` in the repository for example vtscript programs demonstrating the above syntax.

Interpreter Module Specifications

As in project 1, your C++ code implementing the vtscript interpreter must be divided into *at least* the following modules, consisting of a header and implementation pair (.hpp and .cpp). You are free to define additional units of code as desired. Note that the interpreter module itself should use no parts of Qt.

- Expression Module (`expression.hpp`, `expression.cpp`): This module must define a struct or class named `Expression` with at least the following public member functions

```
// Default construct an Expression of type None
Expression();

// Construct an Expression with a single Boolean atom with value
Expression(bool value);

// Construct an Expression with a single Number atom with value
Expression(double value);

// Construct an Expression with a single Symbol atom with value
Expression(const std::string & value);

// Construct an Expression with a single Point atom with value
Expression(std::tuple<double,double> value);

// Construct an Expression with a single Line atom with starting
// point start and ending point end
Expression(std::tuple<double,double> start,
           std::tuple<double,double> end);

// Construct an Expression with a single Arc atom with center
// point center, starting point start, and spanning angle angle in radians
Expression(std::tuple<double,double> center,
           std::tuple<double,double> start,
           double angle);

// Equality operator for two Expressions, two expressions are equal if the have
// the same type, atom value, and number of arguments
// Numbers are compared using absolute value of difference less than machine eps
ilon
bool operator==(const Expression & exp) const noexcept;
```

- Tokenize Module (`tokenize.hpp` , `tokenize.cpp`): This module should define the C++ types and code required to parse a vtscript program into an AST.
- Environment Module (`environment.hpp` , `environment.cpp`): This module should define the C++ types and code required to implement the vtscript environment mapping.
- Interpreter Module (`interpreter.hpp` , `interpreter.cpp`): This module must define a class named "Interpreter`` with at least the following public member functions:

```
// Default construct an Interpreter with the default environment and an empty AS
T
Interpreter();

// Given a vtscript program as a std::istream, attempt to parse into an internal
AST
// return true on success, false on failure
bool parse(std::istream & expression) noexcept;

// Evaluate the current AST and return the resulting Expression
// throws InterpreterSemanticError if a semantic error is encountered
// the exception message string should document the nature of the semantic error
Expression eval();
```

The exception class `InterpreterSemanticError` is provided in the file `interpreter_semantic_error.hpp`.

GUI Module Specifications

The graphical portion of the project should be split into the following modules, consisting of a header and implementation pair (.hpp and .cpp). You are free to define additional units of code as desired, but be sure to use lower-case file names for maximum portability.

- QtInterpreter Module (`qt_interpreter.hpp` , `qt_interpreter.cpp`): This module must define a class QtInterpreter, derived from QObject, that uses compositiopn or inheritance to extend the interpreter module to work in the graphical environment. **It must not simply duplicate the code from the Interpreter Module**. This class should provide at least the following public member functions, signals and public slots.

```
// Default construct an QtInterpreter with the default environment and an empty
AST
 QtInterpreter(QObject * parent = nullptr);

// a signal emitting a graphic to be drawn as a pointer
void drawGraphic(QGraphicsItem * item);

// a signal emitting an informational message
void info(QString message);

// a signal emitting an error message
void error(QString message);

// a public slot that accepts and expression string and parses/evaluates it
void parseAndEvaluate(QString entry);
```

- QGraphicsArcItem Module (`qgraphics_arc_item.hpp` and `qgraphics_arc_item.cpp`): In order to draw arcs according to the specification you will need to customize `QGraphicsEllipseItem` . This module defines a class `QGraphicsArcItem` that publicly inherits from `QGraphicsEllipseItem` and re-implements the paint method to draw an arc without lines from the center to the arc ends.

- MessageWidget Module (`message_widget.hpp` , `message_widget.cpp`): This module must define a class named `MessageWidget` that publicly inherits from QWidget. This class provides a read-only display of interpreter messages. This class should add at least the following public member functions and public slots to the base class:

  ```
  // Default construct a MessageWidget displaying no text
  MessageWidget(QWidget * parent = nullptr);

  // a public slot accepting an informational message to display, clearing any err
  or formatting
  void info(QString message);

  // a public slot accepting an error message to display as selected text highligh
  ted with a red background.
  void error(QString message);
  ```

- CanvasWidget Module (`canvas_widget.hpp` , `canvas_widget.cpp`): This module must define a class named `CanvasWidget` that publicly inherits from QWidget. This class provides a read-only display of graphic items sent to it using a composition of `QGraphicsScene` and `QGraphicsView` . This class should add at least the following public member functions and public slots to the base class:

  ```
  // Default construct a CanvasWidget
  CanvasWidget(QWidget * parent = nullptr);

  // A public slot that accepts a signal in the form of a QGraphicsItem pointer co
  ntaining an
  // object derived from QGraphicsItem to draw
  void addGraphic(QGraphicsItem * item);
  ```

- REPLWidget Module (`repl_widget.hpp` , `repl_widget.cpp`): This module must define a class named `REPLWidget` that publicly inheirits from QWidget. This class provides a way to enter and edit the text of an expression using a QLineEdit widget. It should also implement a history mechanism where the up-arrow and down-arrow key recalls previously entered text is a most-recently-used order. This class should add at least the following public member functions and signals to the base class:

  ```
  // Default construct a REPLWidget
  REPLWidget(QWidget * parent = nullptr);

  // A signal that sends the current edited text as a QString when the return key
  is pressed.
  void lineEntered(QString);
  ```

- MainWindow Module (`main_window.hpp` , `main_window.cpp`): This module must define a class named `MainWindow` that publicly inheirits from QWidget. This class provides the main application interface using a composition of `MessageWidget` , `CanvasWidget` , and `REPLWidget` arranged in a vertical layout. This class need only have two member functions:

```
// Default construct a MainWindow
MainWindow(QWidget * parent = nullptr);

// Default construct a MainWidow, using filename as the script file to attempt t
o preload
MainWindow(std::string filename, QWidget * parent = nullptr);
```

Text-based Interpreter Program Specifications

Note: This section very similar to that in project 1, with the behavior on errors in REPL mode being different.

The interpreter module needs some user interface code. This should be a command-line application that compiles to an executable named `vtscript.exe` on Windows and just `vtscript` on mac/linux. The executable should be usable in one of three ways:

- For executing short simple programs, pass a flag `-e` followed by a quoted string with the program. For example (> is the prompt):

  ```
  > vtscript -e "(+ 1 (- 3) 12)"
  ```

  This should evaluate the program in the string and print the result in the format below or produce an appropriate error message, beginning with "Error", if the program cannot be parsed or encounters a semantic error. If an error occurs vtscript should return `EXIT_FAILURE` from main, otherwise it should return `EXIT_SUCCESS`.
- For executing programs stored in external files, provide the file-name containing the vtscript program as a command-line argument. For example, assuming a file named `mycode.vts` is in the current working directory with the executable:

  ```
  > vtscript mycode.vts
  ```

  This should evaluate the program in the file and print the result in the format below or produce an appropriate error message, beginning with "Error", if the program cannot be parsed or encounters a semantic error. If an error occurs vtscript should return `EXIT_FAILURE` from main, otherwise it should return `EXIT_SUCCESS`.
- For interactive execution of programs using a REPL, just type the executable name:

  ```
  > vtscript
  ```

  This should print a prompt `vtscript>` to standard output and wait for the user to type an expression on standard input. It should then evaluate the provided expression and print the result in the format below, or print an error message, beginning with "Error", if the line cannot be parsed or encounters a semantic error during evaluation. If a semantic error is encountered the environment should **revert to its state prior to the evaluation**. After printing the result the REPL should prompt again. This should continue until the user types the EOF character (Control-k on Windows and Control-d on unix). Changes to the environment should be persistent during the use of the REPL. If the user provides an empty line at the REPL (just types Enter) is should just ignore the input and prompt again.

Output Format: Expressions returned from the interpreter evaluation should be printed as `(<atom>)` for the case Boolean, Number, and Symbol, as `(x,y)` for Point, `((x1,y1),(x2,y2))` for Line, and `((cx,cy),(sx,sy) span)` for Arc. Errors should be printed on a single line as the string "Error: " followed by an error message describing the error.

Example transcripts of use:

- Executing a simple example at the command line:

```
> vtscript -e "(point -2 4)"
(-2,4)
```

- Execute the program in a file (showing it first using cat):

```
> cat program.vts
; print the larger of two numbers
(begin
  (define a 1)
  (define b 2)
  (if (< a b) b a)
)
> vtscript program.vts
(2)
```

- Execute some expressions in the REPL:

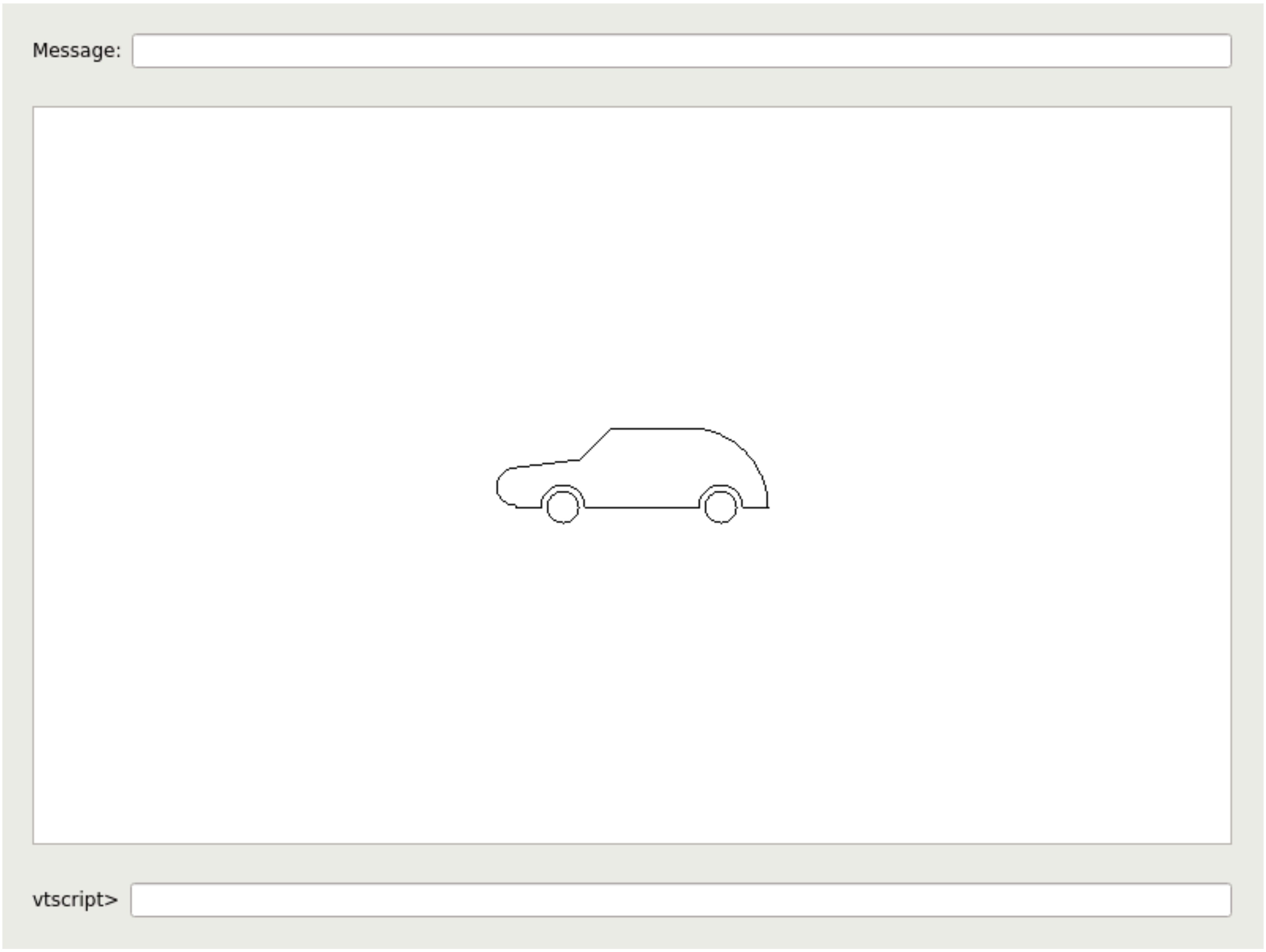
```
> vtscript
vtscript> (define a 12)
(12)
vtscript> (define b 10)
(10)
vtscript> (- a b c)
Error: unknown symbol
vtscript> (- a b)
(2)
```

Graphical Program Specifications

The program vtdraw should be a Qt application implemented in vtdraw.cpp that compiles to an executable named `vtdraw.exe` on Windows and just `vtdraw` on mac/linux. It should instantiate and show the MainWindow class with a minimum size of 800x600, then enter the Qt event loop. This program optionally takes a single command-line argument containing a script file name to read and evaluate, modifying the environment, before any user input can be enetered. If a parsing or semantic error occurs the program should still start but an error message shown in the MwssageWidget.

After starting vtdraw, the user should be able to type expressions into the REPLWidget. Informational messages, such as the resulting expression, should be displayed in the MessageWidget, replacing any existing text. Errors, either syntax or semantic, should also be displayed in the MessageWidget, but the highlight color of the widget should change to a red color and the text should be selected. Expressions that have graphical side effects should appear in the canvas window immediately after they are evaluated. For example running the `test_car.vts` example like so would produce the following window, ready for interaction.

```
> ./vtdraw /vagrant/tests/test_car
```



Unit Testing

For each module of the interpreter code you should have a set of unit tests covering its functionality using Catch as in project 1. Basic tests for the interpreter module are included in the file `test_interpreter.cpp`. These tests should be built as part of your overall project using CMake as described below. These should include your tests from project 1, but be extended to cover the additions to the vtscript language.

For each GUI module above you should have a set of unit tests covering its functionality using the QTest framework. Basic tests for the MainWindow module are included in the file `test_gui.cpp`. You should add tests to this file to test your graphical modules. These tests are built as part of your overall project using CMake as described below.

Using CMake to build and test your software

The repository contains a `CMakeLists.txt` file that sets up the tests I am providing and builds both the vtscript and vtdraw executable. It also creates a configuration header `test_config.hpp` from the file `test_config.hpp.in` when cmake is run so the tests can find the directory containing example vtscript programs (the tests subdirectory).

You will need to modify this file. See the sections at the top marked `EDIT`. In the appropriate section add any source files you create not in the specification for implementing the interpreter, unit-tests, or applications.

Grading Environment

The initial repository includes a Vagrantfile that sets up a virtual machine with minimal graphical interaction enabled. As in earlier projects this is a duplicate of the environment used for grading. This environment adds the ability to interact with graphical applications in a separate window from the shell. After initializing the environment with `vagrant up` you will need to reboot the machine, e.g. `vagrant halt`, followed by `vagrant up` again. This need be done only once (and after any vagrant destroy).

This should allow all parts of the software and tests to be built and run. In the virtual machine this translates to the following:

```
> cd ~
> cmake /vagrant
> make
> make test
```

This treats the source directory as the shared host directory (`/vagrant`) and places the build in the home directory of the virtual machine user (`/home/ubuntu`). Using CMake on your host system will vary slightly by platform and compiler/IDE.

Integration Testing

A basic script is included that tests operation of the vtscript user interface (only). It should be run inside the virtual machine created using the included Vagrantfile. Note it tests for partial correctness only and does not cover all cases. It is intended primarily to ensure our full grading scripts can interact with your vtscript application. In the virtual machine this would typically look like:

```
> cd ~
> cmake /vagrant
> make
> make test
> python3 /vagrant/scripts/integration_test.py
```

To run the vtdraw program start it from the shell. I.e. assuming it is buult in the `/home/ubuntu` directory

```
> cd ~
> ./vtdraw
```

The window should appear in the other VM widow and can be interacted with. To halt the program just type Cntl-C at the shell prompt. On your host machine you can also use the widows decortations or shortcuts provided by your OS (e.g. Cntl-q, Meta-q).

I am providing the binary of my solution. You can install this in the VM using the following:

```
wget https://filebox.ece.vt.edu/~ECE3574/projects/02-vtdraw/VTDRAW_1.0.0.deb
sudo dpkg -i VTDRAW_1.0.0.deb
```

This will add the programs vtscript-clw and vtdraw-clw to your path (installed in /usr/local/bin) which you can run to get the expected output of various expressions.

Submission

Your submission must consist of a git repository containing **only** the source code and any supporting test and configuration files (Vangrantfile, CMakeLists.txt, tests directory and other files provided in the starter repository).

To submit your assignment, either the beta or final versions:

1. Tag the git commit that you wish to be considered for grading as "beta" or "final".

   ```
   git tag <tagname>
   ```

   If you want to have multiple versions of the tags, name them sequentially, i.e. beta, beta2, beta3, final, final2, final3, etc. We will consider the last one present in the repository when it is pulled for grading.
2. Push this change to GitHub

   ```
   git push origin <tagname>
   ```

Be sure you have committed all the changes you intend to by the respective due dates. **Failure to complete these steps by the due date will result in no credit being assigned.**

Grading

There are 100 points allocated to this assignment distributed among the beta and final versions.

	Beta	Final
Correctness	15	35
Testing	10	30
Code Quality	-	10

- The correctness score is determined by the fraction of correctness tests your code passes and whether or not your code has memory errors (leaks, invalid reads/writes, etc) as well as manual verification of testing of the specification.
- The testing score is determined by the quality of your unit tests, that is the amount of coverage they test.
- The code quality score is determined by the code compiling cleanly with no warnings (at the same warning level as P0 and P1) as well as a other criteria that will be provided as part of the beta grading.

Note: Some partial credit may be assigned based on evidence of work and good incremental development practice, as exhibited by your git commit history. Be sure to record your development progress via commits. At the very least this acts as a backup of your work.