Project 1

In which the class of SDVA213 builds a fraction calculator.

This project will be done in four parts, with deliverables at the second and final part.

# Part 1a

Finish by Apr-16

Write classes to operate on fractions.

We need a class to hold a fractional number. It will have a constructor that initializes from another fraction, from integer components, or a string. It will also have a method to convert it to a string. We will implement the “<<” operator on out streams, and the “>>” operator on in streams. It shall also have methods and, as appropriate, operators for each of the following operations:

* Rounding to the nearest 1/n
  + Implement this by multiplying the fraction by “n”, rounding it to an integer, then dividing the result by “n”.
* Addition (“+”)
  + Convert the denominator of both halves to the LCM (Least Common Multiple), then add the numerators.
  + Or, just cross-multiply and add the fraction parts and then normalize as below.
* Subtraction (“-“)
  + Same as addition, only negative.
* Negation (convert a fraction to the negative of itself) (unary “-“)
  + Same as subtraction, only from zero.
* Multiplication (“\*”)
  + Multiply the numerators, multiply the denominators, then normalize.
* Inversion (return a fraction equal to one divided by itself)
  + Just swap the numerator and denominator.
* Division (“/”)
  + Invert and multiply.
* Exponentiation (“^”)
  + (Optional) If the exponent is an integer (denominator = 1), use repeated multiplication.
  + Otherwise: Convert to floating point, then use the “log” and “exp” functions to calculate result, then round it to the original denominator.
* Square root (using Newton’s Method)
  + (Optional) See Wikipedia for an example of this algorithm.

You may add any additional operations that you see fit.

We’re going to store fractions as a negative Boolean and an “improper fraction” to handle values greater than one. After each operation we will re-normalize our fraction by the following steps:

* Calculate the GCD (Greatest Common Divisor) and reduce the fractional part by that multiple.

A small example program using this class follows.

**#include "fraction"**

**int main() {**

**fraction a{11, 4}; // a is 2 3/4**

**fraction b{"3/4"}; // b is 3/4**

**fraction c;**

**c = a + b;**

**cout << c; // prints "3 1/2"**

**c = a \* b;**

**cout << c; // prints "2 1/16"**

**cout << (c + a); // prints "4 13/16"**

**cout << c.round(2); // prints "2"**

**}**

# Part 1b

Due Apr-23

Evaluate expressions involving fractions and print the results.

We’re going to implement simple math with + - \* / operations. While it could be done with just a few functions, I did this as a class called “Calculator”. It uses a recursive descent parser to enforce correct order of operations.

We’re going to describe the language that our calculator accepts in Backus-Noir format (EBNF).

**Expression := [ Variable “=” ] ? Sum.**

**Sum := Product [ [ “+” | “-“ ] Product ]\*.**

**Product := Term [ [ “\*” | “/” ] Term ]\*.**

**Term := Variable | “(“ Sum “)” | Number.**

**Variable := /[a-zA-Z]/.**

**Number := -- The Fraction regex from last week.**

EBNF is a lot like regular expressions, except that instead of matching letters, we’re matching rules. So, for example, a ‘Sum’ is made of a ‘Product’ followed by some number of ‘\*’ or ‘/’, each followed by another ‘Product’. This is the mechanism that helps us calculate ‘4 + 3 \* 2’ as ‘10’ instead of ‘14’. By parsing the characters on either side of the ‘+’ as a ‘Product’, it gives the ‘3 \* 2’ a chance to be calculated before the ‘4 +’ part.

One interesting bit is the ‘Term’. It is basically a single value from the point of view of the enclosing expression. As I read the EBNF it is either a variable, a parenthesized ‘Sum’, or a number. This is what lets us evaluate something like ‘(4 + 3) \* 2’ correctly.

To implement this, we’ll write a parse method for each of these. The various ‘parse’ methods will need to be able to remove characters from the input string once they’ve figured out what to do with them. This is an excellent place to use the ‘regex’ class, this time using ‘regex\_search’ which does not have to match the entire string (as ‘regex\_match’ does). Also, since I prefer to enter things like “a = 1 + (b \* 3/4)” we should be sure to skip any whitespaces that we might find. It turns out that regexes have a ‘\b’ escape which matches whitespace, so that’s handy.

‘parseExpression’ will look at the string that we give it and see if it starts with a variable and an equals sign. If it does, it will pull them off and remember the variable. Then it will call ‘parseSum’ on the rest of the string, and that will give us back a Fraction object. If there was a variable named, we save that Fraction under that variable and either way return it.

‘parseSum’ and ‘parseProduct’ are very similar. They will start by calling some other parse method, then they will look for an operator, and if they find one they will remember it, remove it, and call the parse method again. Once we have the result of each of the parse calls, we can calculate the new Fraction by doing the right computation, simple addition or multiplication or whatever.

‘parseTerm’ is a little different. It can be one of three things. It can start by looking at the start of the string and seeing if it started with some letters. If it does, then go find out what the value for that variable is, and return it. Otherwise, look and see if it starts with a ‘(‘ which means we should parse another sum, then finally make sure there is a ‘)’ and remove it from the rest of the string.

# Part 2a

Finish by May-21

Move our classes into a GUI program where the user can type in an expression and get the result.

For this portion we will put our fraction parser into a Win32 program. Make a windows desktop program like we did in class on May-7, then instead of just doing a temperature conversion make it call the calculate class from Part 1b. Try out the "Project2aSample" for more info.

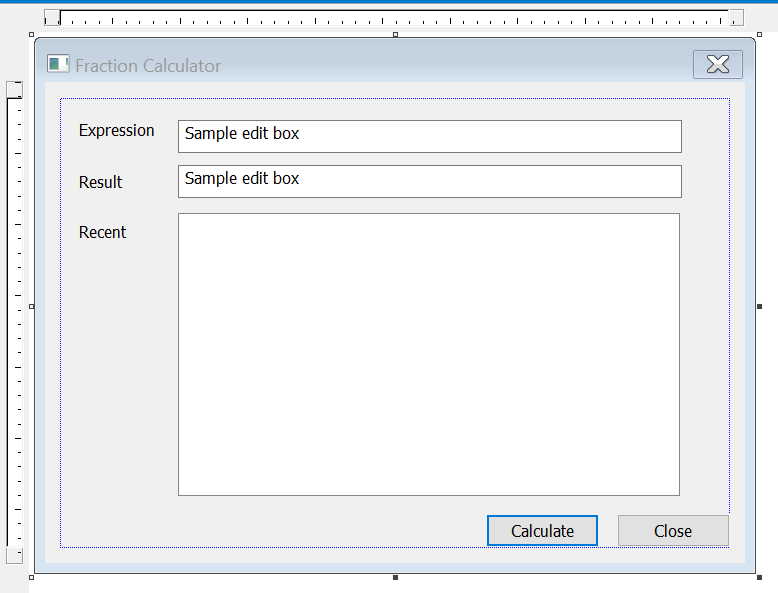
# Part 2b

Due June-11

Build a history window for our calculator. Add a listbox that shows the calculation of all the calculations done in this session.

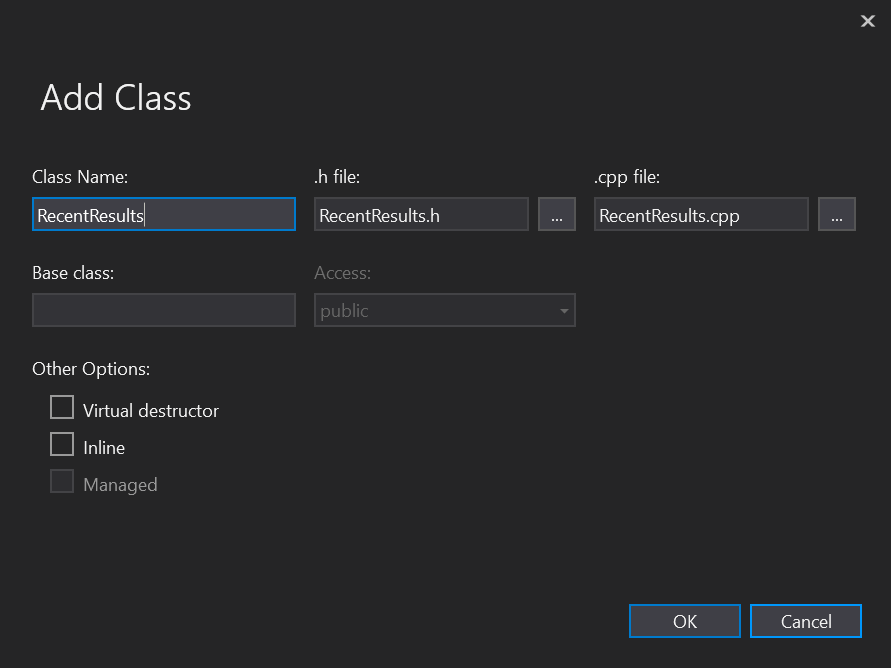
‘List Box’ is a standard component, you can add it to your dialog box from the ‘Toolbox’ in Visual Studio. Figure 1 shows what my calculator’s dialog box looked like after I added a list box to it. The list box will maintain a list of strings for us, and an attached ‘item data’ which is a number. We will use the item data to store the index in an array that will hold the Fraction result.

If we’re going to continue to use a MVC pattern here, we might well next build our Model. I’m going to call mine ‘RecentResults’. If you right-click on ‘Source Files’ then ‘Add’ and ‘Class’ it gives a nice, helpful dialog as shown in Figure 2.



Figure

Of course, you should use a design that makes sense to you, but I decided on giving my ‘RecentResults’ class two methods. The first one is ‘addResult’, which takes a Fraction and returns the index of it in the array. The second one is called ‘operator[]’, which lets me write for example ‘results[0]’ to get the first result. I didn’t add anything for saving the data to a file, or anything like that.



Figure

Now, for the implementation we could be fancy and use our home-made LinkedList class, but in truth it’s almost always better to use the built-in things instead. A ‘std::vector<Fraction>’ does everything we need. Our ‘addResult’ method just calls ‘.push\_back()’, which adds a fraction to the list. Then the ‘operator[]’ method just calls through to the vector’s operator[].

Here’s the interface for my ‘RecentResults’, if that helps:

#include <vector>

#include "Fraction.h"

class RecentResults

{

std::vector<Fraction> recentList;

public:

RecentResults();

int addResult(const Fraction &f);

Fraction operator[](int i);

};

Alright, now we have a View and a Model, so it is time for the Controller. The controller, you might recall, lives in ‘WindowsProject1.cpp’. When we left this, mine had the following code for handling the WM\_COMMAND message:

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case IDOK:

GetDlgItemTextW(hDlg, IDC\_EXPRESSION, text, textLen);

convertResult = WideCharToMultiByte(CP\_UTF8, 0, text, -1, atext, textLen, 0, 0);

try {

result = c->evaluate(atext);

std::ostringstream o;

o << result;

strncpy\_s(atext, o.str().c\_str(), o.str().length());

}

catch (std::string e) {

MessageBoxA(hDlg, ("ERROR: " + e).c\_str(), "Error in expression", MB\_OK);

}

convertResult = MultiByteToWideChar(CP\_UTF8, 0, atext, textLen, text, textLen);

SetDlgItemTextW(hDlg, IDC\_RESULT, text);

return (INT\_PTR)TRUE;

case IDCANCEL:

EndDialog(hDlg, LOWORD(wParam));

return (INT\_PTR)TRUE;

}

The only actions that this responds to are the ‘Ok’ and ‘Cancel’ buttons, which are setup to do the calculation and close the dialog respectively. Inside the ‘IDOK’ one, I’m going to add some extra code that builds up a nice string to put in the list box, then adds it to the list box. Then it will add the Fraction to the ‘RecentResults’ and get back the index in the recent list and set that index in the ‘item data’ for the list box.

Here is the new, streamlined code for displaying the result and building up a nice string to add to the list: (Note that I’m just passing ‘o.str().c\_str()’ directly to MultiByteToWideChar.

o << result;

convertResult = MultiByteToWideChar(CP\_UTF8, 0, o.str().c\_str(), -1, text, textLen);

SetDlgItemTextW(hDlg, IDC\_RESULT, text);

resultIndex = recent.addResult(result);

o = std::ostringstream{};

o << atext << " = " << result;

convertResult = MultiByteToWideChar(CP\_UTF8, 0, o.str().c\_str(), -1, text, textLen);

After this code runs, the ‘text’ variable contains something like “3/8 \* 2 = 3/4”, which we will add to the list.

Now, it turns out that the Win32 API was designed to be used from C as well as C++, so we don’t have anything as easy as ‘list.addItem()’ to work with. Instead of the ‘.’, we have a function called ‘SendMessage’, which is just strange. There’s an example of using a ListBox at <https://msdn.microsoft.com/en-us/library/windows/desktop/hh298365(v=vs.85).aspx>. The sample is old, so ignore the ‘T’ things everywhere. We’ll continue to use the ‘W’ide version of things as we have so far.

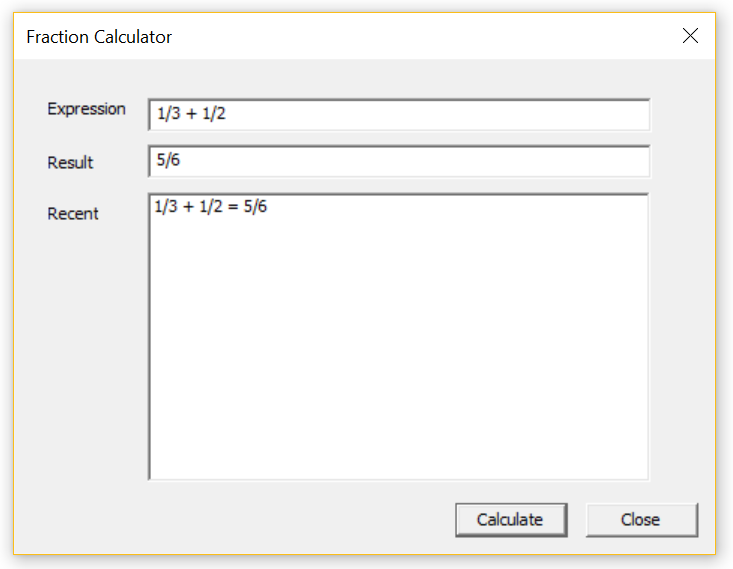
So, looking at the example, we need a ‘hwndList’ to use SendMessage, and we can get that from ‘GetDlgItem’. Remember to use the same ‘IDC\_’ constant that you used when you made the list in the dialog editor. Then you can add the string with something like what I came up with:

listIndex = (int)SendMessage(hList, LB\_ADDSTRING, 0, (LPARAM)text);

We have to remember that listIndex so we can then add the ‘resultIndex’ as the item data.

SendMessage(hList, LB\_SETITEMDATA, listIndex, (LPARAM)resultIndex);

If we run this now, we should see previous expressions added to the list. Clicking on the list doesn’t do anything yet, though, because we don’t have anything in our controller to handle clicks on the list box.



A bunch of the stuff is going to be reused from the IDOK case, so I chose to just copy-n-paste that one and rename the case to IDC\_RECENT. So the flow is, when the user clicks on an entry in the list your window proc will get called with a ‘message’ of ‘WM\_COMMAND’, and in the ‘LOWORD(wParam)’ will be the ID of your list box. I called mine ‘IDC\_RECENT’.

Like before, we need to get the handle to the list by calling ‘GetDlgItem(hDlg, IDC\_RECENT)’. Then we can ask the list for it’s currently selected item with SendMessage.

listIndex = (int)SendMessage(hList, LB\_GETCURSEL, 0, 0);

Now, we have to be a bit careful, because sometimes Windows will tell us that there is no item selected by returning a -1. So, if the listIndex is 0 or greater it means that there is a selection in the list box. Next we want to get the ‘item data’ which is the index in our recent list.

resultIndex = (int)SendMessage(hList, LB\_GETITEMDATA, listIndex, 0);

result = recent[resultIndex];

The rest of the code just about writes itself. We’ll use GetDlgItemTextW and WideCharToMultiByte to get the text out of the ‘expression’ edit box, then use an ‘ostringstream’ to combine the text that was there with the fraction that was selected, and finally use MultiByteToWideChar and SetDlgItemTextW to put the new string back into the edit box. Oh, and it looks nice if you clear the listbox selection by setting the ‘current selection’ to ‘-1’. See the code here for example of how to do that:

std::ostringstream o{};

o << atext << ' ' << result; // o.str().c\_str() contains the new edit box contents

SendMessage(hList, LB\_SETCURSEL, -1, 0); // Clear the selection in the listbox

As usual, start early and please email/text me if you get stuck! Happy coding!