

CS 15 Project 2: CalcYouLater

Contents

Introduction	3
Preamble	3
Note on File I/O	3
Introduction	4
Project Planning and Deliverables	5
Week One	5
Deliverable #1: Design Checkoff	5
Deliverables #2 and #3: DatumStack and parseRString . . .	6
Week Two	6
Deliverable #4: CalcYouLater	6
Datum Class	7
Introduction	7
Datum Interface	7
File Organization for the Provided Datum Files	8
DatumStack Class	10
Introduction	10
DatumStack Interface	10
RPNCalc Class	12
Introduction	12
RPNCalc Interface	12
The ‘Simpler’ RPNCalc Commands	12
The ‘more Complex’ Commands	13
Any rstring	13
exec	14
file	14
if	15

Handling Errors in RPNCalc::run	17
Unrecognized Input	17
Exception Handling	17
Errors while getting command parameters from the stack . .	18
Error output prints to std::cerr	19
parseRString Specification	20
parseRString Implementation	20
Examples	20
Integrating parseRString in Your Code	21
cylc and cyl files	22
Implementation Notes and Advice	23
Notes Regarding the Starter Code	23
Files to Submit	24
DatumStack Implementation	25
Other Tips	25
Getting started	25
Makefile	26
README	26
Submitting Your Work	27

Introduction

Preamble

Welcome to the second project! It's a calculator, but it's even more. It's almost a real programming language! This is a more involved project, for which you have been given ample time, if you start right away.

Do not try to do it in one sitting

Read it right away. Jot down and draw ideas. It's better to work on it in 90 minute or 2 hour chunks of time. If you do that, and you're organized, it will go smoothly. The stack implementation itself should be doable in a single sitting. Be sure to note the three phases of the project. Come in to talk to one of us about your plan as early as you can.

Note on File I/O

For this project, you'll find knowledge you acquired about file I/O in Project 1 helpful. In particular, the ability to pass a reference to an `std::istream` to a function makes it easy to modularize your code by allowing a function to process input from any input stream, whether it be `std::cin` or a stream that you attached to a file using `open`.

There is an additional piece here. C++ has a way for you to treat a `std::string` as if it were an input stream! It has a class, called `std::istringstream`. Its constructor and an example of how to use a `std::istringstream` for input can be found [here](#). A `istringstream` has `eof` and `fail` member functions, just like `std::cin` or any other input stream. Here is also a tutorial on [string streams](#). The idea for our purposes is to make an `std::istringstream` from a `std::string`, and then pass that `std::istringstream` to a function that requests an `std::istream &`, and it will work!

Introduction

In this project you will implement a Reverse Polish Notation (RPN) calculator. RPN is also known as “postfix” notation¹. This means that the operator comes *after* its operands rather than in between. For example:

- $3\ 4\ +$ is equal to 7
- $3\ 4\ *$ is equal to 12
- $1\ 2\ -$ is equal to -1 (note the order)
- $8\ 2\ /$ is equal to 4 (note the order)

Cool fact: RPN does not need any parentheses as long as each operator has a fixed number of operands (which is the case in this assignment). For example, the infix expression:

$$5 + ((1 + 2) * 4) - 3$$

can be written in postfix notation as follows:

$$5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$$

Your `RPNCalc` will support more than just integers, however! It will support `Datum` objects! More on this in a moment.

¹The notation you’re used to, with the operator between the operands is called “infix” notation.

Project Planning and Deliverables

Before writing any functions, sit down and read the assignment specification in full. There is a lot of complexity here! Then, begin to plan your solution. It would be prudent to organize and plan your solution to be as modular as possible. Use helper functions! Doing this initial planning will be extremely helpful down the road when it comes to testing and debugging; it also helps the course staff more easily understand and read your work (which can only help you). It is not advisable that you sit down and attempt to write this in one sitting, particularly if that sitting is close to the deadline. For that reason, the CS 15 course staff has come up with a project implementation plan, split into three **required** (read: graded) phases:

Week One

Deliverable #1: Design Checkoff

Go into office hours and talk to a TA about your plan. Bring some drawings of your data representations - how you imagine your data will flow between structures. This helps twofold: you plan out your project and get your brain working on it in the background, and also you get design feedback before it's too late. You **must** have physical drawings to show the TA. The TA will not look at any code since this checkout is designed to give feedback on your design only, not any code (light pseudocode is allowed). TAs will check off your design and reserve the right to not check off your design if you do not have drawings (and pseudocode) to show or if a TA feels like your design was not thoroughly mapped out enough.

Deliverables #2 and #3: DatumStack class and parseRString

Write and submit the `DatumStack` class and `parseRString` function (these are described in detail under sections of their own names below). These are not expected to be strenuous exercises, but have been known to occasionally hide latent bugs that mess up the rest of your `RPNCalc`. So, start them early, get them right, and make sure they are well-tested.

Week Two**Deliverable #4: CalcYouLater**

Write `CalcYouLater`! (described in detail below) Implement some of the less involved operations first. Then do the more complicated operations.

Datum Class

Introduction

Before diving in to the details of the `RPNCalc` and `DatumStack` classes, we will go over the `Datum` class. A `Datum` object is essentially a container for **one** of three things:

- `int`
- `bool`
- `rstring` (short for RPN string)

The type of the value contained by any given `Datum` object is chosen from the above three options at construction-time, and does not change during the lifetime of that `Datum` object. We have coded the `Datum` class for you, but it would be worth your while to understand the interface.

Datum Interface

- Four Constructors: one constructor for each type a `Datum` object can contain, plus a copy constructor. Note that there is no default constructor. Also note that, under the hood, `rstrings` are represented as C++ `strings`.

```
1 Datum(int i);  
2 Datum(bool b);  
3 Datum(string s);  
4 Datum(const Datum &d);
```

- A destructor
- An assignment operator overload

- Three type query functions:

```
1 bool isInt();  
2 bool isBool();  
3 bool isRString();
```

- An equals (==) operator for comparing two different Datum objects. This allows us to call, e.g., `d1 == d2` where `d1` and `d2` are instances of the `Datum` class. If `d1` and `d2` contain values of different types, this would return false. If `d1` and `d2` contain values of the *same* type, then the underlying values will be compared.
- The less-than (<) operator for `Datum` holding integers. Using the less-than operator and the is-equal-to operator, we can build less-than-or-equal (<=), greater-than-or-equal (>=), and greater-than (>). Use of any of these operators on `Datum` which hold `booleans` or `rstrings` will raise a `std::runtime_error` with the message “datum_not_int”.
- Three data access functions:

```
1 int getInt(); // throws "datum_not_int"  
2 bool getBool(); // throws "datum_not_bool"  
3 std::string getRString(); // throws "datum_not_rstring"
```

These functions each throw an `std::runtime_error` with the associated message above if they are called on a `Datum` of the wrong type. For instance, if I have a `Datum` that contains a `boolean` value, and I call `getRString()` on it, it will raise a `std::runtime_error` with the message `datum_not_rstring`.

- `toString()`, a function that creates a string representation of the `Datum`. This is useful for printing and debugging.

File Organization for the Provided Datum Files

The `Datum` class is given to you as two files: `Datum.h` and `Datum.o`. `Datum.h` contains the interface of the `Datum` class; `Datum.o` contains a pre-compiled object file. That is, it is a non-human-readable file which was compiled from a `.cpp` file, and it contains all of the working machine code that you can use in your project, but does not give you any information about how it works. What it does, however, can be gleaned from `Datum.h` (the interface). To use the `Datum` class, you must `#include "Datum.h"` at the top of whichever file

will use it, and you must link `Datum.o` with your compiled code. You can review the Makefile lecture and lab for a reminder of how to link with a `.o` file.

DatumStack Class

Introduction

The `DatumStack` class will maintain a stack of `Datum` objects, and will be used heavily by the `RPNCalc` class. Part of your Week 1 assignment is to implement this interface.

DatumStack Interface

Your `DatumStack` class must have the following interface (all the following members are public):

- Two constructors as follows:
 - A default constructor, which takes no parameters and initializes an empty stack.
 - A constructor which takes an array of `Datum` and an `integer` specifying the size of the array as parameters and creates a stack initialized so that the elements of the array are on the stack with the array's element 0 pushed on first and it's `(size - 1)`th element at the top of the stack. Example:

```
1 Datum data[2] = { Datum(5), Datum(true) };
2 DatumStack d(data, 2);
3 // d now has a true Datum as the top element
```

- If necessary, define the Big Three (destructor, copy constructor, assignment operator).
- An `isEmpty` function that takes no parameters and returns a `boolean` value that is true if this specific instance of the `DatumStack` class is empty and false otherwise.

- A `clear` function that takes no parameters and has a void return type. It makes the current stack into an empty stack.
- A `size` function that takes no parameters and returns an integer value that is the number of `Datum` elements on the stack.
- A `top` function that takes no parameters and returns the top `Datum` element on the stack. NOTE: It does **not** remove the top element from the stack. If the stack is empty, it throws a `std::runtime_error` exception with the message “`empty_stack`”.
- A `pop` function that takes no parameters and has a void return type. It removes the top element on the stack. NOTE: It does not return the element. If the stack is empty it throws a `std::runtime_error` exception with the message “`empty_stack`”.
- A `push` function that takes a `Datum` element and puts it on the top of the stack.

RPNCalc Class

Introduction

The interface for the `RPNCalc` class is rather straightforward—the complexity is not in the number of functions, but rather is in the logic of processing the commands that come to the `run` function.

RPNCalc Interface

- Define a default constructor which takes no parameters and initializes the `RPNCalc` object.
- Define a destructor that destroys/deletes/recycles any heap-allocated data you may have used in the `RPNCalc` instance.
- Define a `run` function that takes no parameters and returns nothing. This function reads in commands from standard input (`std::cin`). Each command can be read as a string and commands will be separated by whitespace. Commands do not have to be on different lines. See below for details.

The ‘Simpler’ RPNCalc Commands

The supported operations will be extended by the “harder” commands, but implement these simpler commands first and get them working before continuing on.

- A number causes a `Datum` containing the number to be pushed onto the stack.
- `#t` causes a `Datum` with the boolean `true` to be pushed on the stack.

- `#f` causes a `Datum` with the boolean `false` to be pushed on the stack.
- `not` reads and pops the top element off the stack, a boolean, and causes a `Datum` with the opposite boolean value of the popped element to be pushed on the stack.
- `print` prints the value on the top of the stack to `std::cout` (without popping it) followed by a new line.
- `clear` clears the calculator, emptying the stack.
- `drop` causes the top element on the stack to be removed.
- `dup` duplicates the top element on the stack.
- `swap` swaps the top two elements on the stack.
- `quit`, quits the calculator completely. When the program quits, it prints the following message to `std::cerr`: “Thank you for using CalcYouLater.\n” (It should print this message whether it quits with the quit command or by reaching the end of input on `std::cin`).
- The operators `+`, `-`, `*`, `/`, or `mod`. Any of these causes the top two elements (which must both be `integers`) to be popped off the stack, the operation to be performed on them (addition, subtraction, multiplication, division, or remainder), and a `Datum` with the result to be pushed on the top of the stack. The first operand of the operation is the first (deeper) item on the stack. **NOTE: The result does not print.**
- The operators `<`, `>`, `<=`, `>=`, or `==`. Any of these causes the top two elements to be popped off the stack, the operation to be performed on them (some kind of logical comparison) and a `Datum` with the result (a `boolean`) to be pushed on the top of the stack. The first operand of the operation is the first (deeper) item on the stack. **NOTE: The result does not print.**

The ‘more Complex’ Commands

Any `rstring`

You can think of an `rstring` as a sequence of commands to be saved and executed later. For our purposes, an `rstring` will be defined as a sequence of characters that follows this pattern:

- The sequence must begin with "{ " (note the spacing!).
- The sequence must end with " }" (note the spacing!).

Any `rstring` that is provided as input to `RPNCalc` will be put inside of a `Datum` as an `std::string`, and the `Datum` will be pushed onto the stack. To clarify, “`rstring`” itself is not a command; rather, an example of a command that would be parsed as an `rstring` and pushed onto the stack would be “{ 2 8 + }”. Another example would be “{ 2 + }”.

Note: The braces *must* match up, and the spacing around the beginning and ending braces *must* be correct. Also, `rstrings` can be nested - to see an example of nested `rstrings`, read on about if.

Clearly, processing `rstrings` will be important for the success of your `RPNCalc` class. Thus, part of your Week 1 assignment is to implement a function to parse `rstrings`. While reading in input, once you read a "{ ", you should call this parsing function. See the section titled **parseRString Specification** (below) for details.

exec

`exec` takes the topmost element on the stack, which must be an `rstring`, and processes its contents as a sequence of commands. If the topmost element of the stack is not an `rstring`, it should print “Error: cannot execute non rstring\n” to `std::cerr`, and your program should continue to accept input.

file

`file` pops the top element off of the stack, which must be an `rstring`. If it is not an `rstring` it should print “Error: file operand not rstring\n” to `std::cerr`, and continue to accept input.

For example, the `rstring` might be “{ square.cylc }”, in which case the filename is “square.cylc”. The contents of the named file is then read and processed as if its commands had been typed into the command loop. If the file cannot be opened or read, the message “Unable to read FILENAME\n” (where “FILENAME” is replaced with the name of the file specified in the command) is printed to `std::cerr`. The program does not crash or throw an exception. The command loop then continues reading data from its last input source.

if**if Overview**

`if` has a few steps. The command `if`:

1. pops an `rstring` off of the stack—this `rstring` will be executed if the condition is `false`.
2. pops another `rstring` off of the stack—this `rstring` will be executed if the condition is `true`.
3. pops a `boolean` off of the stack—this is the condition to test.

To clarify, `if` assumes that the stack will look like this at the time it is called:

top:	FalseCase	
	TrueCase	
bottom:	TestCondition	

If the test condition is the `boolean true`, then the `TrueCase` in the diagram above should be `exec'd`. If the test is the `boolean false`, the `FalseCase` in the diagram above should be `exec'd`. If any of the elements encountered are of the wrong type, choose the appropriate error message to print to `std::cerr`:

- “Error: expected rstring in if branch\n”
- “Error: expected boolean in if test\n”

Your program should **not** throw an exception. After printing to `std::cerr`, it should continue accepting input.

if Examples

Here are some examples of `if` in action that you might find helpful. Please note that the `>` characters below are the prompt (which you should not output, but we have included for readability in this document), not the greater-than sign.

useless if input to RPNCalc

```
> 3 4 <
> { #t } { #f } if
> print
#t
>
```

translated to C++

```
1 if (3 < 4) {
2     return true;
3 } else {
4     return false;
5 }
```

if that mimics the behavior of “not”

```
> 3 4 <
> { #f } { #t } if
> print
#f
>
```

translated to C++

```
1 if (3 < 4) {
2     return false;
3 } else {
4     return true;
5 }
```

A more complicated nested if

```
> 4 dup 10 ==
> { 1 0 / }
> { 6 + dup 10 < { 10 > } { 10 == } if }
> if
> print
#t
>
```

Which should read as follows:

- Push 4 onto the stack twice. Check if 4 is equal to 10.
- If so, execute the `rstring` “{ 1 0 / }”.
- If not:

- try adding 6 to the 4 on the stack.
- check if that result is smaller than 10 (it shouldn't be).
 - If it is, check if it is greater than 10 (it shouldn't be).
 - If it is not, check if it is equal to 10 (it should be).

translated to C++

```
1  if (4 == 10) {  
2      return 1 / 0;  
3  } else {  
4      if (4 + 6 < 10) {  
5          return 10 > 10;  
6      } else {  
7          return 10 == 10;  
8      }  
9  }
```

Just For Fun Exercises!

- **set** followed by the name of a variable causes the variable to be set to the value on the top of the stack, which is then popped.
- **get** followed by the name of a variable pushes the value of the variable onto the stack. If the variable has not yet been defined, print the variable name and then “: undefined” on `std::cerr` (e.g. “x: undefined”). It should **not** throw an exception. After printing, it should continue accepting input.

Handling Errors in `RPNCalc::run`

Unrecognized Input

For any other input that is not a recognized command, print the offending input and print “: unimplemented” to `std::cerr` followed by a newline (e.g. “x: unimplemented\n”). It should **not** throw an exception. After printing, it should continue accepting input.

Exception Handling

Your `RPNCalc` class should catch any exceptions that may be thrown from the `DatumStack` class. If the `DatumStack` class throws an exception the

RPNCalc should print the string associated with the exception to `std::cerr`. For example if the stack threw an `std::runtime_error` exception with the message “empty_stack”, the RPNCalc should print “Error: empty_stack\n” to `std::cerr`, then continue to accept input.

Note: the exception message from the `DatumStack` class does not include “Error: ”, but RPNCalc should print the error message given when the exception is (or would be) thrown.

Your `run()` function should also catch all other exceptions thrown during evaluation. For example, since the `Datum` class can throw `std::runtime_errors`, you should catch those and print them appropriately to `std::cerr`:

- “Error: datum_not_int\n”
- “Error: datum_not_bool\n”
- “Error: datum_not_rstring\n”

You can review the lecture on Exceptions for a reminder of how to catch exceptions and print the corresponding exception message.

We expect your program to never crash. This means you must test for possible edge cases. If you come up with an edge case, you **MUST** use the reference implementation to determine what the behavior of your program should be for that case, as we will be using the reference, and the exact error messages for grading your assignment.

Errors while getting command parameters from the stack

When running a command that must pop off parameters from the stack, RPNCalc’s `run` should first pop both elements off of the stack, and then display the first error encountered, then continue processing input from the input source. *That means that all parameters that had been popped off by the error producing command will be discarded.*

Example: Datum of wrong type when running the + command

The `+` command expects that the first two elements on the stack are `ints`. If the `+` command was run on a stack that had a `Datum` that wasn’t an `int` at the top, e.g.:

```
top:      | rstring |
          |  int   |
bottom:   | boolean |
          |-----|
```

Then the appropriate error message should be output and the run function should continue processing input. All **Datum** that had popped off by **+** leading up to the error would have been discarded, i.e the stack would look like:

top:		boolean	
bottom:			

Error output prints to `std::cerr`

As mentioned above, all error output (and the final program termination message) is sent to `std::cerr`. The only program output that goes to `std::cout` is generated when `print` command is sent to `RPNCalc`.

parseRString Specification

Introduction and Function Signature

Implementing the `parseRString` function will be required as part of your deliverables for Week 1. Specifically, you will submit a file called `parser.cpp`. It should contain one function called `parseRString` which takes a reference to an `istream`, as a parameter to continue reading input from. The function signature will be as follows

```
1 std::string parseRString(std::istream &input);
```

parseRString Implementation

Assume this function is called after an initial “{” has been read from the input stream passed as a parameter. `parseRString` continues reading input from the stream until all curly braces are matched or until it reaches the end of the input. It returns an `rstring` (i.e, an `std::string`), with the preceding `rstring` specifications.

Examples

```
Example 1: { 1 2 + }
Call parseRString after you see the first {
That is, pass " 1 2 + }" or "1 2 + }" to parseRString
It should return the string "{ 1 2 + }"
```

```
Example 2: { 1 { 4 2 / } / }"
Call parseRString after you see the first {
That is, pass " 1 { 4 2 / } / }" or "1 { 4 2 / } / }"
It should return the string: "{ 1 { 4 2 / } / }"
```

Note: `parseRString` assumes, therefore, that the first “{” has already been processed, and returns a complete `rstring` based on the information after that. If you just call `parseRString` immediately before the first “{”, it will print two “{” characters. This is expected behavior. Later, you will find this function useful when reading in an `rstring` as input—it allows you to read in a *complete* `rstring`, with balanced braces, thereby enabling `RPNCalc` to handle nested `rstrings`.

If `parseRString` gets to the end of the input stream without finding the final matching “}”, then you can choose what to do. Throwing an exception would be reasonable. We will not test you on this case.

Integrating `parseRString` in Your Code

Note that we’ve given you the declaration for `parseRString`, but it’s up to you as to *how* you integrate it into your code for part ii. You have a few options, but we recommend:

- Keep the fn in a file named `parser.cpp` (remove `main.cpp` from this file if it’s there from part i)
- Add the function header to the top of whichever file that uses the `parseRString` function.
- Add a rule to **compile** your `parser.cpp` to `parser.o`
- Make sure to add `parser.o` to the linking step in your `Makefile`.

cylc and cyl files

RPNCalc runs on files with extension `.cylc`. For example, if you have a sequence of commands saved in `my_example.cylc`, you can execute them by either:

- Sending the file to `cin` on the command line by using `<`, i.e., running `./CalcYouLater < my_example.cylc`
- Pushing the `rstring` “{ my_example.cylc }” on to the stack, then using the `file` command.

Often, `.cylc` files can be tricky to read and understand. Thus, you can also use `.cyl` files, which allow you to add comments. In `.cyl` files, comments begin with the character `%`. Because RPNCalc does not support comments, you must then *convert* the `.cyl` file to a `.cylc` file. We have given you a program called `cylc` that does this for you.

As examples, we have provided you with four `.cyl` files in the starter code: `add.cyl` carries out some simple additions, `fact.cyl` implements the factorial function using RPNCalc commands, `fib.cyl` implements the Fibonacci function, and `fib_debug.cyl` uses Fibonacci to demonstrate a way to debug RPNCalc programs. Take a look at these files and the comments within them! They will be a useful reference when it comes time to test your program.

In order to convert the above files to `.cylc` files that can actually be run, use the `cylc` program we have provided you. For example, you can run `./cylc fact.cyl`. This will create a new file called `fact.cylc`, which contains no comments or blank lines, and which can therefore be run by `CalcYouLater`.

Implementation Notes and Advice

You will write both the `DatumStack` and `RPNCalc` classes from scratch. They will each have two files associated with them - a `.h` file and a `.cpp` file. In addition to the public interfaces for the class defined above, you are free to add any private member functions / variables that you wish.

- The names of your functions / methods as well as the order and types of parameters and return types of the interface of the `DatumStack` and `RPNCalc` classes and the `parseRString` functions defined above must be **exactly** as specified. This is important because we will be compiling and linking the code you wrote with our own client code!
- Any print statements or exception messages should likewise print exactly as specified and use the given error type.
- For the two classes, you may not have any other public functions.
- All data members must be private.
- You may **not** use `std::stack` or any other built-in facility that would render the assignment trivial.

Notes Regarding the Starter Code

To get the starter materials, run the following command, which will give you copies of a few files plus links to `Datum.h`, `Datum.o`, and `theCalcYouLater` reference implementation.


```
/comp/15/files/proj2/setup
```

Note - do **not** manually copy the files, just run the command.

- The file `Datum+vector_example.cpp` in the starter code demonstrates the basic usage of the `Datum` class.
- The file, `got_int.cpp`, contains a function `got_int` that can test whether a string can be interpreted as an integer, and, if so, what the value of the integer is. You may use this function as-is: just put it into your file. You should understand how the parameter works, but you do not need to know the details of the implementation.
- The binary file `theCalcYouLater` is a reference implementation that you may refer to to help you understand what you are supposed to do, and to help you test your own implementation.
- The other files are described further in other sections

Files to Submit

The following files for this assignment will be written from scratch:

```
DatumStack.h  
DatumStack.cpp  
RPNCalc.h  
RPNCalc.cpp  
main.cpp  
README  
Makefile
```

You must also submit:

- Any unit tests you write for your program. This may be done in a file called `unit_tests.h`, using the `unit_test` framework that we have used on past assignments. Alternatively, you can create your own testing `main` functions, e.g., submit a file called `DatumStack_tests.cpp` with a `main` function that calls out to tests. Whatever testing files you use, you must submit them!
- Any command files (`.cyl` files that you used for testing, i.e., files that you ran with `RPNCalc` using the `file` command). You should have some command files!

The files `DatumStack.h` and `RPNCalc.h` will contain your class definitions only. The files `DatumStack.cpp` and `RPNCalc.cpp` will contain your implementations. `main.cpp` will contain your main calculator program. The `Makefile` will contain the instructions for make to compile your program. `README` will have the sections described in **README Outline** section below.

DatumStack Implementation

Implement `DatumStack` however you like. We suggest using a `LinkedList` or an `ArrayList` as discussed in class. You may use `std::vector` or `std::list` but you must NOT use `std::stack`.

Other Tips

DO NOT IMPLEMENT EVERYTHING AT ONCE!

DO NOT WRITE CODE IMMEDIATELY!

Before doing anything, use the the reference program to get a sense of how things work. This may seem like a lot, but if you break it into pieces, it's perfectly manageable. Before writing code for any function, draw before and after pictures and write, in English, an algorithm for the function. Only after you've tried this on several examples should you think about coding.

Getting Started

To get up and running,

1. create the two files for the classes `DatumStack` and `RPNCalc`
2. start filling in the `.h` file
3. `#include` the `.h` file in the `.cpp` file
4. define an empty main function in a `main.cpp` file (just return 0)
5. compile!

If you start with the `DatumStack` class (follow a similar pattern if you start with the `RPNCalc` class):

1. Implement just the default constructor first. Add a single variable of type `DatumStack` to your test main function; compile, link, and run.

2. Now you have some choices. You could add the destructor next, but certainly you should add a print function for debugging soon. Make it private before submitting.

You will add one function, then write code in your test file that performs one or more tests for that function. Write a function, test a function, write a function, test function... Follow the same testing approach for every class you write, and add functionality little by little.

For the `RPNCalc` class, the first version of the `run()` function should accept only the `quit` command. Then it should read any number of items, just printing them, until there's a `quit` or until reaching end of file. Then start to add items bit by bit. Add numbers, and print the stack to make sure they get on the stack correctly. Then add the operators one at a time, comparing your results with the reference. Do the `file` command last.

You may want to refactor your program when you get to the `file` command; don't duplicate the command logic in two places. If you pass file streams around, you should do so using C++ reference parameters. The example program we provide shows how.

If you need help, TAs will ask about your testing plan and ask to see what tests you have written. They will likely ask you to comment out the most recent (failing) tests and ask you to demonstrate your previous tests.

Makefile

We have given you a Makefile that knows how to make `.cylc` files from `.cyl` files, but you must fill in the rest. Your `CalcYouLater` program must build when we run `make CalcYouLater`, and must produce an executable named "CalcYouLater" which can be run by typing `./CalcYouLater`.

README

With your code files you will also submit a `README` file. You can format your `README` however you like. However it should have the following sections:

- A The title of the homework and the author's name (you)
- B The purpose of the program
- C Acknowledgements for any help you received

- D The files that you provided and a short description of what each file is and its purpose
- E How to compile and run your program
- F An outline of the data structures and algorithms that you used. Given that this is a data structures class, you need to always discuss the the data structure that you used and justify why you used it. Please discuss the features of the stack data structure and also include (with some justification/explanation) two other situations/circumstances/problems where you could utilize a stack. The algorithm overview is always relevant. Please pick a couple interesting/complex algorithms to discuss in the README.
- G Details and an explanation of how you tested the various parts of assignment and the program as a whole. You may reference the testing files that you submitted to aid in your explanation.

Each of the sections should be clearly delineated and begin with a section heading describing the content of the section. You should not only have the section letter used in the outline above.

Submitting Your Work

Be sure your files have header comments, and that those header comments include your name, the assignment, the date, and acknowledgements for any help you received (if not already credited in the README file).

For phase 0, there is nothing to provide, but you must have your design reviewed by a member of the course staff. Come in to office hours and see someone. Bring your design, which should include pictures of your data structures, lists of functions you will need, etc.

For phase 1, the command is:

```
provide comp15 proj2_calculator_phase1 DatumStack.h \
                                     DatumStack.cpp parser.cpp README \
                                     unit_tests.h ...
```

Don't forget to include your `rstring` parser code. The README doesn't have to be the final README. Just document anything that you feel we should know about your `DatumStack`, `parser`, or the project in general.

For the final submission, the command is:

```
provide comp15 proj2_calcyoulater DatumStack.h DatumStack.cpp \  
    unit_tests.h RPNCalc.h \  
    RPNCalc.cpp main.cpp README Makefile ...
```

where “...” means any other files necessary to build your program as well as any test code (e. g., .cyl files) you would like us to see. For example, if you have another class, you should name the .h and .cpp files for that class there.

CalcYouLater!