1 Project 2 Big Bits

Due: Oct 3 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then provides some background information. It then lists the requirements as explicitly as possible. It lists the files with which you have been provided and describes how those files are organized into modules. It considers tradeoffs between different design possibilities. It then provides some hints as to how those requirements can be met. Finally, it describes how to submit the project for grading.

1.1 Aims

The aims of this project are to expose you to:

- Abstract data types.
- The use of pointers in C.
- Postfix expressions.
- The C API for dynamic memory allocation.

1.2 Background

This section reviews material on multi-precision integers, postfix expressions and a short-hand notation for inputting large repetitive hexadecimal numbers.

1.2.1 Multi-Precision Integers

C provides operations on standard integer types `int`'s, `long`'s, and `long long`'s. Even the last guarantees only 64 bits, so if bigger integers are needed it is necessary to use some kind of bigint package like the *GNU Multiprecision Arithmetic Library* `gmp` which allows the use of integers of size limited only by available memory. Such a general package would be extremely full featured and support highly optimized arithmetic and bitwise operations on a wide variety of multiprecision numeric types.

This project only requires you to implement some of the simplest operations: the bitwise operations, with minimal optimization.

1.2.2 Postfix Expressions

In order to test your project, you are being provided with a command-line program which works as a simple calculator for bitwise operations. This calculator allows the input of bitwise expressions using **postfix notation**. Hence in order to test your project, you will need to understand *postfix notation*; the next few paragraphs provide a overview.

The standard notation for arithmetic expressions uses *infix notation* where an operator is written in between its operands, as in `4 + 2 * 5`. One problem with this notation is that it is ambiguous and requires additional information about the associativity and precedence of the operators in order to disambiguate. For example, in the absence of knowledge about associativity and precedence, `4 + 2 * 5` could be interpreted as either `(4 + 2) * 5` or as `4 + (2 * 5)`; we normally interpret it as `4 + (2 * 5)` only because of the rule that `*` has precedence over `+`. If the former interpretation is desired with the standard precedence rule, then it is necessary to use explicit parentheses as in `(4 + 2) * 5`.

In postfix notation, the operator is written after its operands, and it is possible to write an expression unambiguously without needing any parentheses. For example the infix expression `4 + (2 * 5)` can be written in postfix as `4 2 5 * +`, while the infix expression `(4 + 2) * 5` can be written as the postfix expression `4 2 + 5 *`.

Evaluating of postfix expressions is particularly simple, which is why it was chosen as the syntax for the bitwise calculator for this project. The evaluation uses a *last-in first-out LIFO* stack. The algorithm for evaluation scans the "words" in the expression left-to-right, proceeding as follows:

1. Initialize the input to the words in the postfix expression and the stack to an empty stack.
2. If the leftmost word in the input corresponds to an operand, push it on to the stack.
3. If the leftmost word in the input corresponds to an operator, then pop the 2 topmost operands off the stack, operate on them using the operator and push the result on to the stack. It is an error if there are not at least two entries on the stack to pop.
4. Remove the leftmost word in the input.
5. If there are words left in the input, then go back to step (2).
6. The value of the expression is on top of the stack. If there is more than one expression in the stack, then we are missing an operator.

For example, the evaluation of the postfix expression `4 2 + 5 *` would proceed as follows (the stack is shown with its "top" on the right):

Input	**Stack**
4 2 + 5 *	[]
2 + 5 *	[4]
+ 5 *	[4, 2]
5 *	[6]
*	[6, 5]
	[30]

OTOH, the evaluation of the postfix expression `4 2 5 * +` would proceed as follows:

Input	**Stack**
4 2 5 * +	[]
2 5 * +	[4]
5 * +	[4, 2]
* +	[4, 2, 5]
+	[4, 10]
	[14]

The examples above involved arithmetic expressions. The project uses involves bitwise expressions involving hexadecimal operands and postfix bitwise operators `|`, `&` and `^`.

1.2.3 Easy Input of Long Hexadecimal Integers

Since this project deals with big bits, testing is facilitated by having some easy way to input large hexadecimal integers. Hence the provided command-line program allows a compact representation of hexadecimal numbers having a prefix repeated some number of times.

Specifically, a hexadecimal literal can can contain a `*` followed by a single positive hexadecimal character specifying a repetition count for the prefix of the literal seen so far. Examples (underscores are merely used to aid readability):

- `0xf*4` represents `0xffff`.
- `0x2a*a` represents `0x2a2a_2a2a_2a2a_2a2a_2a2a`.
- `0x3*4*2` represents `0x3333_3333`.
- `0xab*2c*2` represents `0xab_abca_babc`. This can be analyzed as follows: `0xab*2` represents `0xabab`; `0xab*2c` represents `0xa_babc` and the final repetition by 2 in `0xab*2c*2` represents `0xab_abca_babc`.

1.3 Requirements

Submit a `submit/prj2-sol` folder in your `i220`X repository in github such that typing `make` within that folder produces a `big-bits` executable within that directory which when run enters an interactive postfix bitwise calculator with interaction similar to that shown in this log.

The input and output for the program as well as the postfix evaluation algorithm is implemented by code which you are already being provided with. What you need to write is an implementation of the big-bits ADT specified in big-bits.h.

There is no requirement that your implementation is highly optimized, but it should meet the following additional requirements:

- There should not be any restriction on the size of a big-bits integer beyond that imposed by the amount of available memory.
- If every big-bits created by calling `newBigBits()`, `andBigBits()`, `orBigBits()` or `xorBigBits()` is freed up by calling `freeBigBits()`, then all allocated memory should be cleaned up.

1.4 Provided Files

The provided prj2-sol directory includes the following files:

big-bits.c A skeleton file which you will need to modify.

big-bits.h A specification header file which provides the specification you need to implement. You should not modify this file.

hex-util.c and hex-util.h Some utility functions for dealing with hexadecimal representations of integers. You may find some of these functions useful in your code.

main.c This provides the postfix bitwise calculator. You should not need to modify this file.

stack.c and stack.h Specification and implementation of a stack ADT which is used by the postfix calculator. You should not need to use this module directly, but you can use it as an example of how an ADT can be implemented when you are implementing your bit-bits ADT.

errors.c and errors.h Specification and implementation of a trivial error reporting module. It provides two functions `error()` and `fatal()` where each function takes `printf()`-style arguments and prints the message specified by its arguments on standard error; the difference between the two is that `fatal()` additionally terminates the program. You should not need to modify these files.

You should not need to use the functions provided in these files as a module should depend on its client for error reporting. This module is used by the command-line postfix calculator program.

Makefile A `Makefile` with default target which builds the entire program. It also provides a `clean` target for cleaning out object and executable files and emacs backup files. You should not need to modify this file.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

Additionally, the aux-files directory contains the following files:

Interactive log file big-bits.LOG A log of an interactive session with the postfix calculator.

Test files The input data big-bits_test1.in for the above log and a file containing the corresponding output big-bits_test1.out. These files can be used for non-interactive *regression testing* which is much less tedious than testing interactively:

```
$ PRJ2_AUX=$HOME/cs220/projects/prj2/aux-files
$ ./big-bits <$PRJ2_AUX/big-bits_test1.in >t.out
$ diff $PRJ2_AUX/big-bits_test1.out t.out
```

If the tests are successfully, the `diff` command should complete silently; if they fail, then it will show the difference between the expected and actual output.

1.5 Design Considerations

You should design your implementation of the big-bits ADT before you start coding.

It is important to clearly distinguish different concepts and use clear and consistent names for them:

Hex Char A character represented by a C `char` `'0'` ... `'9'`, `'a'` ... `'f'` or `'A'` ... `'F'`.

Hexet A value between `0` ... `15` representing the integer represented by a hex char.

Nybble or Nibble A representation of a hexet when packed into a memory word, often two per byte.

An obvious implementation for the ADT is to represent a big-bits by multiple occurrences of a **unit** representing some C integer primitive. A bitwise operation for big-bits would involve iterating that operation over all the units in the big-bits representation. Some choices for the unit:

`char` A big-bits is represented by a sequence of multiple bytes. A choice in this representation is whether each byte should contain only a single nybble or two nybbles. The former is simpler but less efficient. In the former case, one nybble is wasted per byte; in the latter case, at most one nybble would be wasted per big-bits representation.

`short`, `int`, `long`, `long long` In all these cases, multiple nybbles would be packed into a unit and a big-bits would be represented by a sequence of `short`, `int`, `long`, `long long` depending on the unit chosen. The advantage of using these bigger units is that bitwise operations for big-bits will require fewer iterations. A disadvantage is that the representation for each big-bits could waste up to `2 * sizeof(unit) - 1` nybbles and that the code for dealing with the most significant unit would be complex.

C merely guarantees a minimum size for these types. Hence the code cannot assume a fixed number of nybbles per unit if it is to be portable; instead, it will need to calculate that number dynamically.

The `BigBits` structure would merely contain a dynamically allocated array of units and track the number of units in the array.

Once you decide on a unit for your representation, you will need to decide on endian issues:

- Are the units in the representation of a big-bits stored little-endian or big-endian? A little-endian representation makes the bitwise operations easy, but the conversion to-and-from hex strings harder. A big-endian representation makes the conversion to-and-from hex strings easier, but the bitwise operations harder.
- Are the nybbles within a unit stored big-endian or little-endian. This should not make any difference in implementing the bitwise operators as you can operate on all the nybbles within a unit using a single operation. It could however make a difference in the complexity of converting to-and-from hex strings.

Another design decision is whether your representations should be normalized to contain the fewest number of units needed to represent the value of the big-bits.

- A normalized representation of a postfix expression like `0xf*a*a 0xf*a*a ^` would contain only a single unit containing `0`, whereas an unnormalized representation could contain up to 100 zero nybbles.
- A normalized representation may make the code for converting the representation back to a hex string easier but the code for the bitwise operations may become more complex.

- An unnormalized representation may make the code for converting the representation back to a hex string somewhat harder but the code for the bitwise operations may become simpler.

1.6 Hints

My implementation currently clocks in at 185 lines including all the boilerplate comments and declarations included in the provided *skeleton file*.

With the representation I chose, the functions for converting between my representation and hex strings were relatively complex. Hence it was not possible to easily test my code until these functions were implemented completed. This did not allow incremental development of those functions interspersed with testing. The flip side was the code for doing the bitwise operations turned out to be extremely straight-forward.

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Review all relevant material, specifically the material ADTS in detail and the *C Introduction* in general.

2. Come up with a design for your representation, based on the tradeoffs outlined earlier.

3. Make sure your local `cs220` repository is up-to-date:

   ```
   # assumes you have set up your account as directed
   $ cd ~/cs220
   $ git pull
   ```

4. Copy over the provided files to your `work` folder in your `i220X` local repository:

   ```
   # also assumes you have set up your account as directed
   $ cd ~/i220X/work  #X will be a or b
   $ cp -pr ~/cs220/projects/prj2/prj2-sol .
   $ cd prj2-sol
   ```

5. Fill in the README template.

6. Implement the `newBigBits()` function. Some of the functions in the provided hex-util module may be useful.

   It is important to realize that the `0x` prefix, ignored `_` characters and the `*`-repeat feature within hexadecimal literals are handled entirely by the provided code; your ADT implementation needs to only handle pure hexadecimal strings containing only hex chars.

7. Implement the `freeBigBits()` function. This should be trivial.

8. Implement the `stringBigBits()` function. Depending on whether your representation is normalized, you may have to do some work to ensure that your returned hex string does not contain any non-significant zeros.

   At this point, you should be able to start testing your implementation with the provided postfix calculator; simply typing a hexadecimal representation into the calculator should round-trip the hex string into your ADT representation and back out as a hex string which is displayed by the calculator.

9. Implement the `andBigBits()` function. This should be relatively straightforward. All you will need to is iterate the `&`-bitwise operation for the number of units contained in the shorter operand.

10. Implement the `orBigBits()` function. This too should be relatively straightforward. In this case you will need to is iterate the `|`-bitwise operation for the number of units contained in the longer operand, making sure that the higher-order units which are not present in the shorter operand are treated as `0`.

11. Implement the `xorBigBits()` function. This should be similar to the `orBigBits()` function.

12. Iterate until you meet all requirements.

13. Add any relevant information to the README.

1.7 Submission

When you are happy with your project, move it over from your work directory to your submit directory:

```
$ cd ~/i220X #X is either a or b
$ git mv work/prj2-sol submit
$ git commit -a -m 'suitable comment'
$ git push
```

This should submit your project as `submit/prj2-sol` via github. Your submission should not include any object files or executables; this will be prevented by the provided *.gitignore* file.

If submitting late, please drop an email to the TA for your section:

Section A yli241@binghamton.edu

Section B rrausha1@binghamton.edu