Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2013
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays and structures, and functions that make use of them. You will also extend your skills in terms of program design, testing, and debugging.

The Story...

Most modern cryptography relies on doing arithmetic with very large numbers. One example is the RSA algorithm, used by secure websites to prove that the server really is who it's claiming to be. RSA can also be used for encrypting messages. Either way, its security relies on the assumption that it's hard to factorize a number that is the product of two large primes, where "large" means hundreds of decimal digits.

The problem is, the data type `int` in C can only store values up to $2^{31} - 1 = 2{,}147{,}483{,}647$. If larger values must be manipulated accurately, a different representation for integers is required. Your task in this project is to develop an *integer calculator* that works with extended-precision integer values. The calculator has a simple interface, and 26 "variables" (or memories) into which values can be stored. For example a session with your calculator (where > is the prompt from the system) might look like:

```
mac: ./ass1
> a = 2147483647
> a ?
2147483647
> a + 1
> a ?
2147483648
> b = 999999999999999
> c = 1000000000000000000
> c + b
> c ?
1000999999999999999
> b ?
999999999999999
> a ?
2147483648
> ^D
mac:
```

Note the extremely limited *syntax*: constants or other variables can be assigned to variables using an "=" operator; or variable values can be altered using a single operator and another variable or constant (and "+" is the only one implemented in the skeleton; or variable values can be printed using "?". The variable in question is always the first thing specified on each input line; then a single operator; and then (for most operators) either another variable name, or an extended-precision integer constant.

To allow storage of very large integers, your calculator needs to make use arrays of ASCII characters, with one decimal digit stored per character in the array. You are to design a suitable structure and

representation, assuming that values of up to `INTSIZE` digits are to be handled. Other information might also be required in each "number", including a count of the number of digits, and a sign.

The expected input format for all stages is identical – a sequence of simple calculator commands of the type shown above.

Stage 1 – Getting Started (marks up to 8/15)

Obtain a copy of the skeleton file using the link on the LMS, and spend some time (hours!) studying the way it is constructed, including compiling and executing it. It only works with `int` variables at present, but is already quite a complex program!

Note the use of the type `longint_t`. Each of the `longint_t` variables in the array `vars` stores one integer, with 26 variables available in total, named `'a'` to `'z'` (and indexed from 0 to 25, via the function `to_varnum`). These variables are changed by various functions, depending on the commands read from the input. There is no subtraction, but negative numbers are available via negative constants:

```
> a = 10
> a + -20
> a ?
-10
```

In this first stage you are to design and implement an alternative data structure for `longint_t`. You should use an array of `INTSIZE` characters for each `longint_t`, together with the other required information, all bound together as a `struct` (Chapter 8).

You will need to make non-trivial changes in many of the functions (and perhaps introduce new ones), and need to be highly systematic in the way you approach this task. You should plan carefully, rather than just leaping in and starting to edit the file. Then, before moving through the rest of the Stages, you should test your program carefully, to make sure that you have retained all of the existing functionality. And remember that if you truly scramble things up, you can always recopy the skeleton and start again.

Stage 2 – User Friendly Input and Output (marks up to 10/15)

Alter the program so that numbers are always printed with commas (Australian-style, every three digits), and can also (optionally) have (correctly or incorrectly placed) commas in them when they are entered:

```
> a = 2147483647
> a ?
2,147,483,647
> b = 2,147,483,647
> b ?
2,147,483,647
> c = -21,47483647
> c ?
-2,147,483,647
```

Stage 3 – Another Operator (marks up to 15/15)

Now add in a multiplication operator, `*`:

```
> a = 12,345,654,321
> a * 1,234,321
> a ?
15,238,500,387,151,041
```

You will need to think carefully about the *algorithm* that you will use in this task. Best bet is to implement some functions that will allow you to carry out the calculation the way that you learned to do long multiplication at school. For example, a function `times_ten` that multiplies a number by ten might be useful; as might a function `times_digit`, which takes a `longint_t` and multiplies it by a single-digit number.

There are also other ways (some of them much more efficient than the "school" algorithm) to doing multiplication, and you can be inventive if you wish. Just don't try and do it by repeated addition.

Stage 4 – Yet More Operators (marks up to 15/15)

Of course, you know what is coming next – an integer "power of" operator, "^":

```
> a = 2
> a ^ 100
> a ?
> 1,267,650,600,228,229,401,496,703,205,376
```

If you are successful in this stage, you will be able to earn back one or two of the marks you lost in the earlier stages (assuming that you lost some, you can't go past 15/15 in total). But *please please please* don't start on this task until your program through to Stage 3 is (in your opinion) perfect, and is safely submitted, and is verified, and is backed up somewhere. (Still keen? Add in division too, using "/".)

The boring stuff...

This project is worth 15% of your final mark. You don't have to do Stage 4 in order to get 15/15.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the LMS once submissions are opened. You will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. Only the last submission that you make before the deadline will be marked.

You may discuss your design/plans during your workshop, and with others in the class, but what gets typed into your program must be individual work, not from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say "**are you out of your mind?**" if they ask for a copy of, or to see, your program, pointing out that your refusal, and their acceptance of it, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.*

Deadline: Programs not submitted by **6:00pm on Friday 20 September** will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email Alistair, `ammoffat@unimelb.edu.au`.

And remember, algorithms are fun!