School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2023
Assignment 1

Learning Outcomes

In this assignment you will demonstrate your understanding of arrays, strings, functions, and the `typedef` facility. You may also (but are not required to) use `structs` (see Chapter 8) if you wish. You must *not* make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project.

Big Numbers

It is sometimes necessary to work with very big numbers. For example, application areas such as cryptography require exact manipulation of integers containing hundreds of digits. And by now you will be aware that the standard `int` and `double` data types do not possess that ability.

In this project you will develop a high-precision numeric calculator. You'll be using C arrays to store the digits of the numbers, so it won't quite be arbitrary precision; but it will certainly be possible to have 500-digit values manipulated exactly within reasonable computation times, which is big enough for many purposes.

Before doing anything else, you should copy the skeleton program `ass1-skel.c` from the LMS Assignment 1 page, and spend an hour or two reading through the code, understanding how it fits together. Check that you can compile it via either `Grok` or a terminal shell and `gcc`. Once you have it compiled, try this sequence:

```
mac: ./ass1-skel
> a=12345
> a+23456
> a?
register a: 35801
> a+a
> a?
register a: 71602
> ^D
ta daa!!!
```

The "`>`"s are a prompt printed by the program, with the remainder of each of those lines the commands typed by the user. There are also two output lines in response to the "`?`" commands, and the final (what else?!) "`ta daa!!!`" message printed by the program as it exits. (Note that there is some fiddly code that makes use of the `isatty()` function to decide where to write each output line, so that the program works sensibly when input commands come from a file, and also when the output is directed to another text file. You do not need to understand how all that code works.)

The calculator maintains 26 one-letter "variables" (or *registers*), each of which has an initial value of zero, and can store one integer value. Each "command" operates on one of those 26 variables, applying a single operator to it, using (except for the "`?`" operator) one further non-negative integer operand. So, in the example shown above, register "`a`" is first assigned the value 12345; then it has 23456 added to it; then it is printed; and then register "`a`" has register "`a`" added to it; and then it is printed a second time.

The skeleton program uses a simple array of `int` variables to store the 26 registers. This skeleton is provided as a starting point, so that you don't have to develop a whole lot of uninteresting functions.

The only operators provided in the skeleton program are "=" (assignment); "+" (addition); and "?" (printing). And the arithmetic will suffer from integer overflow, of course.

Stage 1 – Find Your Own Type (12/20 marks)

The first change in the program is to employ a new definition for the type `longint_t` so that it becomes an array of `INTSIZE` decimal digits, each stored as an `int` in an array, plus a matching buddy variable stored in the first array position. That is, each register will be an array of `int` (rather than a single `int`), with the digits stored in *reverse* order, and with the first element in the array storing the array's buddy variable. (You'll understand why the digits are to be stored in reverse order once you start coding your solution.) For example, the number $12{,}345{,}542$ has eight digits, and would be stored in a variable `longint_t v` as:

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]	...
8	2	4	5	5	4	3	2	1	?	?	...

where `v[0]` stores the (current) number of digits in the register's value; where "?" represents a value that isn't currently in use; and where the array `v` is declared to contain a total of `INTSIZE+1` elements. All input numbers will be non-negative integers, and nor is a subtraction operator required. That means that you do not need to store a sign, nor worry about negative numbers.

If you wish to read Chapter 8 early, you may instead define and use a suitable `struct` for each of the registers (rather than the array option that is illustrated in the example), and then maintain the set of registers in an array of struct (rather than as an array of arrays). Note that the use of `struct` types is *not* required in this project, and you can get full marks without them.

As part of this first stage you'll need to rewrite the function `do_plus()` so that two variables of your new type `longint_t` are added correctly, and you'll also need to modify several other functions (`zero_vars()` and `do_assign()`, plus others) so that they operate correctly after you have changed the underlying type. You should carefully plan the required changes *before* you jump in and start typing, because you'll need to do quite a bit of editing before getting the program back to "compilable/testable" state again.

As part of your changes, your program should check for overflow beyond `INTSIZE` digits both when constants are being input, and also when addition is being performed. If an overflow situation arises your program should print a suitable error message (your choice of wording) and then exit.

Successful completion of this stage means that your program should be able to handle additions involving numbers of up to `INTSIZE` (currently 500) decimal digits. Your program should detect any operations that overflow beyond `INTSIZE` digits, and write an error message of your choice and then exit, returning `EXIT_FAILURE`.

Stage 2 – Go Forth and Multiply (16/20 marks)

You surely knew it was coming, right? Well, now is the time to add a multiplication operator:

```
> b=123456789
> b*987654321
> b?
register b: 121,932,631,112,635,269
```

Yes, you need to sit down with pen and paper and remember your long multiplications from primary school, and then turn that process into a C function in your program. If you get stuck, ask your parents (or grandparents!), they might still remember how to do this kind of multiplication.

You should *not* implement multiplication via repeated addition, that will be frighteningly inefficient. At the other end of the spectrum, nor are you required to investigate efficient integer multiplication algorithms. It is expected that your process for multiplying two n-digit numbers will involve time proportional to n^2, that is, will be $O(n^2)$.

As a second change required in this stage, note the commas in the output values, following the Australian convention of placing them every three digits, counting from the right. (Other cultures have different conventions.)

Stage 3a – Seize the Power (19/20 marks)

And what comes after multiplication? Exponentiation, of course.

```
> c=2
> c^50
> c?
register c: 1,125,899,906,842,624
```

Just as multiplication shouldn't be done via repeated addition, exponentiation shouldn't really be done by repeated multiplication – it isn't an efficient algorithm. But in the context of this project you may carry out exponentiation via repeated multiplication (and that is certainly what you should get working first) since the largest (interesting, non-overflowing) exponent that can arise within 500-digit numbers (that is, generating values less than 10^{500}) is $\log_2 10^{500} = 1661$, and executing a few thousand many-digit multiplications should still only take a tiny fraction of a second. Have fun!

Stage 3b – Divide, and then Conquer (20/20 marks)

[*Note: This last operator involves a great deal of further work for one measly little mark. Think carefully about your other subjects and assessment priorities before embarking on this stage, because it might become a black hole that consumes a lot of your precious time. There is no shame at all in submitting a program that handles Stages 1 to 3a only; and for many (perhaps even most) of you, that will be the correct decision to make.*]

For an exciting adventure, add in integer division:

```
> d=b
> d/c
> d?
register d: 108
```

with `b` and `c` having the values assigned in the two previous examples. Hopefully your grandparents haven't forgotten how to do long division and can teach it to you (and I bet your parent; or look at `https://en.wikipedia.org/wiki/Long_division`. You may *not* implement division via repeated subtraction, but probably will need some subtraction-based looping for each digit in the quotient that is generated.

General Tips...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

The sequence of stages described in this handout is deliberate – it represents a sensible path through to the final program. You can, of course, ignore the advice and try and write final program in a single effort, without developing it incrementally and testing it in phases. You might even get away with it, this time and at this somewhat limited scale, and develop a program that works. But in general, one of the key things that makes some people better at programming than others is the ability to see a design path through simple programs, to more comprehensive programs, to final programs, that keeps the complexity under control at all times. That is one of the skills this subject is intended to

teach you. And if you submit each of the stages as you complete it, you'll know that you are accumulating evidence should you need to demonstrate your progress in the event of a special consideration application becoming necessary.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 15 September**.

Submissions that are made after the deadline will incur 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 `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks and feedback will be provided approximately two weeks after submissions close.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; 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 a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The skeleton program includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.

And remember, algorithms are fun!