

Practical 2: I'm the operator...

Instructions

This is an assessed practical in five parts, A–E. Each part of the practical guides you through the construction of part of a program.

To obtain credit for your work, you will need to submit electronically the most advanced version of your program.

The deadline for completion and electronic submission of Practical Exercise 2 is 16:00, Friday 14th November (week 8). In the absence of evidence of medical or other difficulties, work signed for or submitted after the deadline will attract reduced (possibly zero) credit.

Aims

- To provide experience in tackling a somewhat larger problem than any of those encountered in *Practical 1*. This larger problem will be solved by breaking it into simpler subproblems, whose size is comparable to those we saw in *Practical 1*.
- To provide practice in defining and using function in C. These functions will encapsulate solutions to the subproblems just mentioned.
- To provide experience in dealing with textual input, albeit in a slightly sanitised environment.

Assessment

The maximum credit that can be obtained for this practical is 35 marks (out of a total of 100 for the whole of the continuously assessed element of the course). The five parts, labelled A–E, differ in value: Part A is worth 15 marks, Part B 4 marks, Parts C and D 6 marks each, and Part E 4 marks. Try to complete at least Parts A–C, but don't spend *excessive* time on the practical, particularly Parts D and E. Note that a high quality solution just to Parts A–C may achieve a B grade.

Notes

- Read through this document *before* you reach the keyboard, and work out in advance what you need to do.
- If you are genuinely stuck, seek help from your tutor or the course lecturer.

Electronic Submission

Unless you decide to attempt Part E, there is just one program file to be submitted, namely `calc.c`. The specific form of the command for submitting this file is

```
submit cs1 cp1 P2 calc.c
```

The version you submit should be the most advanced one that works. It's a good idea to cover yourself by submitting your current version of `calc.c` at the end of each part. This is fine, since newer versions simply overwrite the older ones.

In addition you are invited to submit a summary file thus

```
submit cs1 cp1 P2 summary
```

in which you have *briefly* set out:

- Which parts of the exercise you consider you have successfully completed.
- Any problems you encountered in the parts you attempted but didn't complete.
- Any features of your program that you wish to draw attention to.

This should be a short document. Try to keep it to say a dozen lines.

The input format for Part E varies slightly from the other parts. Therefore, if you make an attempt at Part E you should submit it as a separate file `partE.c`, in addition to `calc.c`. Obviously the form for this will be

```
submit cs1 cp1 P2 partE.c
```

Preparation

This practical has associated template files. You will need to copy these files into your directory so that you can modify them.

To copy the templates, make sure you are in the directory that you have created for this practical and issue the command

```
cp /group/teaching/cp1/Prac2/* ./
```

Introduction

The aim of the exercise is to build a simple calculator.

Part A [15 marks]

In this part we'll tackle the case of a single addition with two operands, for example `1+2`. Note that that form of the input is very restricted: just single digit numbers and no spaces! It's quite a long part, but you'll be well on the way to a pass mark when you get to the end.

Dealing with textual input is a surprisingly tricky business, so we'll make life a little easier by using some predefined functions (and constants) described in `io.h`. Here is a summary of what is provided.

- C regards 1 as true and 0 as false. For the benefit of the human reader, `TRUE` and `FALSE` are declared as constants with exactly these values.
- There is an input buffer whose contents may be displayed at any time by making the function call `PrintInputBuffer()`. The symbol `@` is used to mark where we are now; characters to the left of `@` have already been read, while those to the right are yet to be read.
- `ReadSymbol()` returns the next character from the input buffer.
- `ParseError("Your error message")` writes "Your error message", and then displays the input buffer.
- Sometimes it is only after we have read a character that we realise we have gone too far. (C.f. the problem of getting off a bus at the closest stop in an unfamiliar city!) `UnReadSymbol(ch)` places the character `ch`, presumably the one we just read, back in the input buffer.
- The function `ReadNumber` is actually in the file `calc.c` and not `io.c`. The intention is that `ReadNumber(&n)` should read a non-negative number from the input buffer and place it in the integer variable `n` (and return true if successful). Currently it does this... but only if the number is a single digit.

That in brief is the raw material; for further details, consult the file `io.h`.

Now take a look at the file `calc.c`. What do you think this program does? Test your intuition by compiling the program (`make calc`) and running it.

Recall that our goal is to recognise and evaluate simple addition sums such as `1+2`. To this end, you are required to write a function `ReadExpression` with the same format as `ReadNumber`:

```
int ReadExpression(int *value)
```

(The fact that `ReadExpression` and `ReadNumber` have the same format will be significant if you reach the final part of the exercise.) The aim of `ReadExpression` is to read an expression of the form: `number+number`, where `number` denotes (for the time being) a single digit number. Use the functions `ReadNumber` and `ReadSymbol` to achieve this goal. Return `TRUE` only if the whole expression is

read successfully. In this case, assign the sum of the two numbers just read to the call-by-reference parameter `value`. Otherwise, return `FALSE`, leave `value` unchanged and leave the input buffer in any state.

Tip: During development, use debugging aids such as `PrintInputBuffer()` and `printf("check point 1")`, etc, liberally, to follow the processing of the input buffer. Remove these debugging statements before submitting the program.

Test `ReadExpression`. Ensure that appropriate error messages are reported, using `ParseError`, when the input expression contains errors.

Part B [4 marks]

Write a function

```
void SkipWhiteSpace(void)
```

to skip over any sequence of space (' '), newline ('\n') or tab ('\t') characters at the current point on the input buffer. The input buffer should be left at a point immediately after the white space..., which will probably involve calling `UnReadSymbol()`. Test `SkipWhiteSpace()` using `PrintInputBuffer()`.

Modify `ReadExpression()` (and/or the functions it calls) to allow expressions containing space to be read, e.g., `\n 7 + 8 \n`. You face a design decision here: where is the best place to use `SkipWhiteSpace()`? Different solutions will work, but some may be cleaner than others.

Part C [6 marks]

Modify `ReadExpression` to work with the operators `+`, `-`, `*`, and `/`, rather than `+` alone. Start by writing a function

```
int ReadOperator(int *operator)
```

`ReadOperator(&op)` should return true only if one of the following operators is read: `+`, `-`, `*`, and `/`. The call-by-reference parameter to `ReadOperator` (in this case `op`) should be assigned the character value of the operator returned.

Write a function

```
int ApplyOperator(int operator, int left, int right).
```

Depending on the value of `op`, the function call `ApplyOperator(op, x, y)` should return the sum, difference, product or (integer) quotient of the operands `x` and `y`. Since the operator parameter will presumably be provided by the function `ReadOperator` we are not expecting any errors when `ApplyOperator` is called.

Now test `ReadExpression` with input expressions using all of the four operators above: e.g., `5*7`, `8 - \n3`, `2\t*\t3` and `7/2`.

At this point, you have completed the core of the exercise. You can get a safe pass mark, or maybe even a good one, by submitting a good solution to Parts A–C. Part E is conceptually more difficult than Part D, but Part D involves writing more code. If you complete Part E, you will have successfully written a recursive function, even if you didn't notice your achievement at the time!

Part D [6 marks]

The function `ReadNumber` currently reads a number between 0 and 9. For this part, you are required to modify `ReadNumber` so that larger numbers can be read, e.g., 594 or 7654321. To obtain full credit, `ReadNumber` should be restricted to read numbers with a maximum of 7 digits. `ReadNumber` should make the function call `ParseError("Number too large")` if this condition is violated. `ReadNumber` should return true if a number is successfully read and false otherwise.

Note: after `ReadNumber` is called, the current position in the input buffer (i.e., the `@` in `PrintInputBuffer`'s report) should be at a point immediately after the whole number. E.g.,

```
Buffer: 5@\n
Buffer: @ten\n
Buffer: 55@\n
```

Test your function with at least the following values.

```
5 0 9 ten # 7654321 87654321
```

Part E [4 marks]

After all this hard work, this is where the program suddenly becomes interestingly complex. Because the input format changes slightly in this part, you should create a copy of `calc.c` in a file `partE.c` and work with that. You wouldn't want to lose your earlier working program if Part E doesn't work out for you! Note that you may compile `partE.c` by issuing the command `make partE`.

At this point, you should have thoroughly tested your program and you should be happy that it works and that it leaves the buffer in the correct state at each point.

You currently have a calculator that recognises inputs of the form

$$\textit{expression} := \textit{number operator number}$$

By modifying `ReadExpression`, extend the calculator to cope with more general inputs, which are described by the grammar

$$\textit{expression} := \textit{number} \mid (\textit{expression operator expression})$$

We'll be meeting formal grammars later in the course, but for the time being, think of this formula as saying: “An expression is *either* a number, *or* a left-parenthesis, followed by an expression, followed by an operator, followed by a second expression and finished with right-parenthesis”. (This *seems* suspiciously circular, but don't lose heart!) So that we're sure we all have the same notion in mind, here are some examples of expressions:

```
2
(5 * 9)
(1 + ((5 + 4)*2))
```

The values of these expressions should be 2, 45 and 19, respectively. Notice that we insist that expressions are “completely bracketed”: this means that the order of evaluation is explicitly defined and avoids the problem of *parsing* the expression.

It seems that `ReadExpression` will need to call itself. Will that work? Try it!

Checklist

- Electronically submit the program `calc.c` and the brief **summary** by 16:00, Friday 14th November (week 8).
- If you make an attempt at part Part E, please submit that too in a file `partE.c`

(practical by) James Soutter and Mark Jerrum.

*I am adding and subtracting
 I'm controlling and composing
 By pressing down a special key
 It plays a little melody
 By pressing down a special key
 It plays a little melody*

*I'm the operator with my pocket calculator
 I'm the operator with my pocket calculator*

The above lines are *not* © Edinburgh University.