目录

Week 6 Exercises

sumdig.c

A C executable treats its command-line arguments as strings. Write a program that sums all the digits (0 ... 9) that appear in all its command-line arguments.

- all characters in the arguments that are not digits should be ignored
- if there are no arguments, the program does nothing

Possible executions are:

```
prompt$ ./sumdig
prompt$ ./sumdig 12 34
10
prompt$ ./sumdig 1u1u 1u11aby
5
prompt$ ./sumdig 1 --2-- ++3++
6
prompt$ ./sumdig 000410002003
10
prompt$ ./sumdig @^*+{}~?+-?8,.=^%[]@^*+{}~?+-?8,.=^%[]
16
```

ackermann.c

Write a program that computes Ackermann's function, given by:

```
A(0, n):= n+1 for n>=0
A(m, 0):= A(m-1, 1) for m>0
A(m, n):= A(m-1, A(m, n - 1)) for m>0, n>0
```

The program should read 2 non-negative integers *m* and *n* from the command line, compute Ackermann's function for these numbers, and print the result.

If the number of command-line arguments is incorrect, or an argument is not an integer, then a *Usage* message is printed. If an argument is negative, then an error message that says that the function is undefined for negative integers should be printed.

Examples of executions are:

```
prompt$ ./a.out
Usage: ./a.out m n
prompt$ ./a.out 1 2 3
```

```
Usage: ./a.out m n
prompt$ ./a.out 1 a
Usage: ./a.out m n
prompt$ ./a.out -1 0
Ackermanns function is not defined for negative integers
prompt$
prompt$ ./a.out 2 1
Ackermann(2, 1) = 5
prompt$ ./a.out 4 0
Ackermann(4, 0) = 13
prompt$ ./a.out 4 1
Ackermann(4, 1) = 65533
```

Ackermann(4,1) took about 4 minutes to compute on the CSE network computer *williams* by the way, using the compiler optimisation *-O3*.

array2heap.c

Before you commence this exercise, do the following:

- by hand, make a heap from the numbers **1, 2, 3, 4, 5, 6, 7**
  - do this by inserting each element in turn into the growing heap, starting with an empty heap
- when finished, flatten the structure so that you can see what the heap looks like as an array

Now the programming exercise. Assume you have an array of integers:

```
切换行号显示

1 int heap[] = {-999,1,2,3,4,5,6,7};
```

hard-coded into a program. In spite if its name, this array clearly is not a heap.

Convert the above array into a heap *in-place*, and print the result. Note the following:

- there is no input in this program
- *in-place* means you change the elements in the array itself: you do not need any other data structures
- use the function *fixUp* from lectures (it may need modification)
- remember, the root element in a heap is at location 1. I've set location 0 to -999 in the array above to indicate it is not used.
- when you print the final heap, include the zero location just to confirm it is still -999

args2heap.c

We now want to take out the hard-coded array in the previous exercise, and instead use the command line.

Write a program that reads integers on the command line, builds a heap from the integers as they are being read, and prints the heap when complete. So, for example:

```
prompt$ ./a.out 1 2 3
-999 3 1 2
```

If there are no arguments, or any argument is non-numerical, a *Usage* message is printed.

```
prompt$ ./a.out 1 2a 3
Usage: ./a.out integers ...
prompt$ ./a.out
Usage: ./a.out integers ...
prompt$ ./a.out 1 2 abc
Usage: ./a.out integers ...
```

Note the following:

- do not use arrays (except of course for *argv[]*, which is needed to read the numbers)
    - (this means you will need to use *malloc()* to store the heap)
- do not forget to put a '-999' at the zeroth position in the heap
- you should build the heap as you read each integer
- when you have processed all the command-line arguments, print the final heap
- make sure your program is leak-free and there are no dangling pointers

Run the test case from the previous exercise to check it's working:

```
prompt$ ./a.out 1 2 3 4 5 6 7
```

which should produce the same heap as the previous exercise.

cHeap.c

In this exercise we go back to characters.

Write a program that reads single, alphanumeric characters on the command line and stores them in a heap. The heap is printed if it is non-empty. The zeroth character in the heap is printed as '#'.

The program should ignore (i.e. throw away) any argument that is not a single alphanumeric character. The alphanumeric characters set is [0-9A-Za-z] by the way.

Examples of use:

```
prompt$ ./a.out red rover move over
prompt$ ./a.out @ [ }
prompt$ ./a.out
prompt$ ./a.out a b c
# c a b
prompt$ ./a.out abcd ef gh a b c
# c a b
prompt$ ./a.out 1 e g g
# g g e 1
prompt$ ./a.out 1 A a 2 B b 3 C c 4 D d END
# d b c C D a 3 1 B 2 4 A
```

Notice in the *1 e g g* execution that the heap has lower-case 'a' at the top, above the upper-case characters. This is because the character comparison in *fixUp()* (the condition in the *while* loop) actually compares ASCII codes, and lower-case letters have ASCII codes > ASCII codes for upper-case letters (which in turn are > ASCII code for digits).

A reminder that the command ***ascii*** on the CSE network, or any Linux/unix computer will show the ASCII codes of all characters. You might also like to look at ***ctype.h*** on the C Reference Card for help in checking whether a character is alphanumeric.