Lab Exercise 6: The Pangolins Game

Duration: 2 sessions

Aims

To check that you have learnt enough C to be able to cope with the algorithms part of this course-unit in the second semester.

Learning outcomes

On successful completion of this exercise, a student will:

- Have written a C program to create and manipulate binary trees, and to read and write from files
- Have used `valgrind` to perform simple checks on the program

Summary

Write a C program `pangolins.c` that plays the game of Pangolins (described below). As well as this lab script, you are provided with:

- a longer description of the game of Pangolins with examples,
- examples of the use of a tree to play the game, and
- hints about how to code the game in C.

This exercise is split into 4 parts of increasing difficulty, with most of the marks awarded for the first 3 parts. You should try to complete the first two parts by the end of the first lab session, and the next two parts by the end of the second lab session.

Deadlines: The unextended deadline is the end of your second scheduled lab for this exercise. If you need it, if you attend the lab you will get an automatic extension to the start of the next COMP26120 lab session (you must use submit to prove you finished in time and get it marked at the start of your next scheduled lab - the marking session in Week 12). You can also get an extension for good reason e.g. medical problems.

Description

For this lab exercise you should do all your work in your `COMP26120/ex6` directory.

Copy the `makefile` into your `COMP26120/ex6` directory from `/opt/info/courses/COMP26120/problems/ex6` (use this path).

Debugging: *When you are using pointers, there is every chance that at some point your program will crash with the message:* `Segmentation fault`
If you can't work out what happened, try using a debugger (e.g. `man gdb` *or* `man ddd`*).*
e.g. to use `ddd`*:*

- *Run ddd, loading the program that is crashing e.g.* `ddd pangolins`
- *Click "Run" on the buttons, or Program->Run*
- *If you get a pop-up window for arguments etc., just click Run*
  The top pane will use a big red arrow to point to just before the line of code where the program crashed. The bottom pane will give more detailed information e.g. about function parameters and the line number.
- *Click "Up" on the buttons, or Status->Up*
  The top pane will use a big grey arrow to point to just before the line of code where the previous

function was called. Again, the bottom pane will give more detailed information. Repeat this step if you need to.

I hope this will help give you some idea where to look for the problem. If you can't see it, then at least you know where to start putting `printf` statements.

The Game Of Pangolins

Pangolins is a two player game, played between you and your computer. The idea is very simple. You think of something (an animal, a type of pudding, some form of fermented bean-based product, anything you like) and the computer tries to guess what you are thinking of by asking you a series of 'yes/no' questions. If the computer guesses correctly, it wins. If you manage to fool it, you win, but you must then provide the computer with a question that would allow it to guess correctly next time round. That's all there is to it.

Here's an example game. Your input is prefixed by '>', and the computer's output is in bold text.

```
Does it have a tail?
> No
Is it flat, round and edible?
> Yes
Is it a pizza?
> No
Oh. Well you win then :-(
What were you thinking of?
> a biscuit
Please give me a question about a biscuit, so I can tell
the difference between a biscuit and a pizza
> Can you dip it in your tea?
What is the answer for a biscuit?
> Yes
Thanks
```

The game is straightforward. The computer starts knowing only about a single 'object' (i.e. a thing, nothing to do with object oriented programming). Whatever object you're thinking of, the computer will offer its first and only possible answer as its guess. If this guess turns out to be wrong, the computer will ask you for three pieces of information:

1. The name of the object you were actually thinking of
2. A question (with a yes/no answer) that distinguishes between the computer's recent guess and your object
3. The answer to your question in the case of your object - yes or no.

The computer accumulates this information in a very naive way, so it can use it in subsequent rounds of the game. For example, there is no attempt to improve the order of the questions or to stop the player giving nonsense or inconsistent information. But that's okay; it's just a C programming exercise, not an attempt to create artificial intelligence.

A longer example game of Pangolins, with several rounds

How does the Pangolins game work?

Part 1: Basic structures

You will first need to design the data structures. In C you are limited to 'structs' (rather than the true 'objects' which Java provides) but there's nothing wrong (and quite a lot good) about thinking of these things in broadly object-orientated terms. However you decide to arrange the details of your data structures, they will form a binary tree, with objects and questions at the nodes and 'yes/no' decisions at the edges.

Step 1A: Define your data structure e.g. using a struct. You might also use a union and an enum. (Hints about data structures.)

Step 1B: Write a diagnostic function called `nodePrint` that will print out the status of a single node of the tree. (This should help you with debugging.)

Look at the decision tree at the end of round 5 of the example game. If we called the function `nodePrint` on the "does it like to chase mice" question, we might expect output something like this:

```
Object: [NOTHING]
Question: Does it like to chase mice?
Yes: A cat
No: a pangolin
```

Alternatively, calling the same function on the "a pizza" node would give output something like:

```
Object: a pizza
Question: [NOTHING]
```

Notice in this second example we have not printed out the 'Yes' and 'No' alternatives to the question since there isn't a question.

Step 1C: Now, you should 'hard-code' a small tree of questions and answers into your program: you'll need this both to test your `nodePrint` function and later to get the game off to a good start. We suggest you play the game on a piece of paper, and build up a small tree (say 6 or 7 nodes) of questions and objects. Now create a data structure to represent this, and add it to your program. No need to use dynamically memory at this stage - you can just create the data structures using statically declared variables.

Step 1D: Now, write a new function - `treePrint` - which prints out the whole game tree you have just hard-coded. Parts of `treePrint` will, of course, be similar to `nodePrint`, but it will use recursion to print out all the nodes of the tree in a systematic way. You can get rid of `nodePrint` when you are happy that your `treePrint` is working correctly. (Hints about printing a tree.)

Part 2: Making the game interactive

Now you have the basic data structures in place, we'll put the rest of the game in place. This is relatively straightforward, and involves simply using `fgets` to get input from the user and then building new parts of the tree in response to what the user says.

When you hard-coded the data structures in part 1, they were conveniently allocated on the stack for you, strings and all. Since the compiler could tell from looking at your program how long your questions were, how many questions you had, how long the object names were, and so on, it could allocate appropriate amounts of stack for the job. However, when the game is being played 'live', there is no possible way of predicting these sizes, so you will have to use dynamic allocation techniques to store the strings that are typed in by the user, as well as the structs that form the tree itself.

The game should allow for arbitrarily long questions and object names, and in principle the tree can be extended indefinitely. This means that it is simply not possible to use static arrays to store the tree or its contents; you'll have to dynamically allocate memory using `malloc`.

One exception to this open-endedness, is that in order to read a string of characters from the user, it must be stored somewhere; there is no 'standard' function in C that will allow you to read in a string of unbounded length since you have to pass a buffer to the functions into which the string will be copied (so you have to know how long your maximum string/buffer size is in advance.) Unless you want to go to the effort of writing an extensible input-getting function, this means you have to choose a sensible length of buffer to statically allocate. Use one of the standard 'input' functions from the C library (we'd recommend `fgets`) making sure to tell the function what the maximum number of characters to accept is, and finally copy the string from your fixed size static buffer into a snug-fitting dynamically allocated buffer within your game tree. It's not a perfect solution, but it will do fine for us here. (Hints about performing input.)

A neat way of packaging this slightly unsatisfactory state of affairs up would be to write a function that asks for a user input of a defined maximum size, and then returns a `malloc`-ed string exactly fitting the string that the user actually entered.

Once you have a function (or functions) to deal with user input like this, coding the logic of the game so that appropriate questions are asked, appropriate guesses made and the tree dynamically created is relatively straightforward. (Hints about the basic logic of the game.)

To get full marks for this part, you should make your game robust to the types of input that a user might give. For example: if the user gives 'pangolin' rather than 'a pangolin' as a reply, does you program produce text that reads correctly the next time round? (i.e. does it assume that the user has given the correct article ('an', 'a', 'the' etc.) before the object name?) What happens if the user forgets to put a question mark at the end of their question? Can they type 'y' as well as 'yes'? Could they even type 'absolutely' or 'correct'? You won't be able to mitigate against all possible types of nonsense input, but given that the user is not trying to test an artificial intelligence program but might make a few genuine mistakes, can your program cope gracefully with these?

This would be a good place to get to by the end of the first lab session.

Part 3: loading and saving the game

Of course, at the moment, every time you finish your program, it will forget about everything it has learned. Hardly what you'd want. Your task now is to write functions to load and save the question tree to a file.

Step 3A: Save your game tree to a text file. (Hints about saving and loading the tree.)

Step 3B: Read a saved game tree and recreate it in memory. Play the game to verify that the tree is correct.

To simplify testing, you should probably use program parameters to set the filenames, and only initialise and/or save the tree if filenames are given.

Part 4: Tidy up your mess

When your program exits, all the resources such as file handles and memory chunks that you have used get returned to the operating system. This is a good thing, because most programmers leave huge piles of messy pointers and lumps of partly eaten memory lying about when their program finishes. For this exercise, you are to be much tidier.

When you program is ready to finish, make sure you have explicitly `free`-ed all the memory that has been `malloc`-ed by you. You might want to do this recursively.

Use `valgrind` to convince yourself that there is nothing going awry and that all the tiniest bits of memory have been `free`-ed correctly (you will need to demonstrate `valgrind` giving your program a clean bill of health to get your marks for this task).

Marking Process

You must use `labprint` and `submit` as normal.

`labprint` and `submit` will look for: `pangolins.c`

This lab gets tougher as it goes along, and the marking scheme is kept quite tight and to-the-point. Once more there are a few marks allocated for 'subjective' issues like style, tidiness and clarity of code, and a few marks for really quite difficult bits towards the end. The majority of the marks are awarded on a 'does it work' basis.

The marks are awarded as follows:

```
1 mark : Part 1A - a sensible data structure
1 mark : Part 1C - nodePrint function
3 marks: Part 1D - treePrint function
3 marks: Part 2  - a basic working game of pangolins
3 marks: Part 2  - dealing sensibly with user input
3 marks: Part 3A - saving your game to a text file
3 marks: Part 3B - loading your game back from the file
```

```
3 marks: Part 4 -  using valgrind to show that all your data
                   structures are tidied up nicely before your program exits

Total 20
```