COMP26120 Lab Exercise 4
Arrays and Memory Management

Duration: 1 session

Aims

To encourage you to find out more about the use of pointers and data structures, and about some of the memory management features of the C language.

Useful C on-line course themes etc.:

- Differences between Java and C
- Lots of information about Structs, Unions, Pointers and Linked Lists in Information Representation

Learning outcomes

On successful completion of this exercise, a student will:

- Know how to use structs, unions, pointers, malloc and free in C;
- Be aware of some fundamental differences between C and Java - the explicit use of pointers and of memory management;
- Know how to use valgrind to help debug C programs.

Summary

Use C to write a series of programs that repeatedly call:
`insert (structure, name, age, ...)`
to put information into 1-dimensional Arrays (Vectors)

1. insert at end of array of struct
2. insert at end of array of pointer to struct, and using malloc
3. using free and valgrind
4. pointer (reference) parameter

The unextended deadline is the end of your scheduled lab.
If you need it, if you attend the lab you will get an automatic extension to the start of your 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).
You can also get an extension for good reason e.g. medical problems.

You will continue using your program in lab exercise 5, so make sure you leave it in a good state.

Debugging

To debug a program, I usually insert `printf` statements (remember to output a **newline** `"...\n"` at the end or even use **`fprintf(stderr,...)`** otherwise you might not see incomplete output generated just before a crash) but when you are using pointers there is every chance that your program will behave very strangely or even crash with the message: `Segmentation fault`

`valgrind`
: is well suited to identifying problems with pointers or `malloc` (even if your program seems to be working correctly).

debuggers like `ddd` or `gdb`
are well suited to identifying where a program is crashing.
Some people also prefer to use a debugger to identify less dramatic problems, rather than inserting `printf` statements.

Use `man` to get some information about how to use these programs.

valgrind

`man valgrind` says it is "a suite of tools for debugging and profiling programs"

The simplest way to use it is without the `--tool` option, to just check the use of pointers and the heap. This tends to produce a lot of output that can be hard to understand at first, so **use it before you have a problem** to see what the output looks like when everything is ok.

Moreover, `valgrind` can spot potential problems even if they aren't (yet) serious enough to cause your program to crash, so **get into the habit of using it whenever you use pointers and malloc**.

ddd

`man ddd` says:

> "DDD is a graphical front-end for GDB and other command-line debuggers. Using DDD, you can see what is going on "inside" another program while it executes--or what another program was doing at the moment it crashed.
>
> DDD can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:
> - Start your program, specifying anything that might affect its behavior.
> - Make your program stop on specified conditions.
> - Examine what has happened, when your program has stopped.
> - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another."

For example, to find where a program is crashing:

- Run ddd, loading the program that is crashing e.g. `ddd arrays`
- 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.

For more information about `ddd`, try `ddd --manual` (here) or Norm Matloff's ddd Tutorial.

Description

For this lab exercise you should do all your work in your COMP26120/ex4.

Copy the starting files from `/opt/info/courses/COMP26120/problems/ex4` These files are: `arrays.c` and `makefile`

You should write a single C program, `arrays.c`, for all parts of this exercise. You are given an initial version of this program, as mentioned above. You should probably keep a working back-up each time you progress to a new step.

The parts become harder as you progress. If you are running out of time, don't try to complete all parts - instead get everything marked that you can by the deadline, and try to prepare better for the next lab exercise. (Make sure that you use submit to prove that you finished in time.)

Part 1: Array of Struct

Edit the program to:

- Declare a `struct` (type) that describes a person. It should contain a string for a name and an int for an age (in years).
- Declare an array of these structs (called `people`).
- Complete the `insert` function to insert a name and age into the next unused element in the arrray. Use a `static` variable (e.g. called `nextinsert`) inside the function to remember where the next unused element is. (I'm not proud of this, but it does work, and you will get rid of it later.)
- Inside `main`:
  - In a `for` loop, call `insert` to put the next name and age into the `people` array.
  - Use a **second** loop to print the contents of the `people` array.

It is important that you pass the complete array of structs to `insert`, not just one element of it.

Part 2: Array of Pointer to Struct

Edit your program so that your array `people` is now an array of **pointers** to structs. Modify `insert` to call `malloc` to create a new struct and set the correct array element pointing to it. Remember to check the result of `malloc` for errors (and you can test this by temporarily giving `malloc` a ridiculously big number as a parameter).

Part 3: Tidying up using `free` and `valgrind`

Edit your program so that, after the array has been printed in `main`, you use a **third** loop to call `free` to release the memory allocated by `malloc`.

If you haven't already done so, find out about the `valgrind` command (e.g. using the section about debugging above and `man valgrind`) and use it to check that you have got this part right (e.g. `valgrind arrays`).

Part 4: A Pointer (or "ref" or "var") Parameter

- Move the declaration of `nextinsert` from inside `insert` to inside `main` (i.e. following the declaration of your array `people`).
  Get rid of the `static` modifier on the declaration, but keep the initialisation to 0.
- Add `nextinsert` as a parameter to the declaration of `insert` (i.e. `static void insert(..., int nextinsert)`) and to the call from `main` (i.e. `insert(..., nextinsert);`).

Compile and run your program as normal. It should fail, with only one person in your array at index 0, and probably a "Segmentation fault" as it hasn't set up the other array items. (If you haven't used a debugger yet, get some practice using it now, to find out which line your program crashed at.)

The problem is that, although `insert` increments the parameter `nextinsert`, the changed value doesn't get returned from the function at the end of the call, so `nextinsert` in `main` never changes and each new person is being put into the first array item (i.e. at `[0]`).

To fix this, we could try to use the return result from the function, but instead we are going to make the parameter a **reference** (i.e. **pointer**) to the variable whose value we want to modify. This is similar to the way `sscanf` puts input values into variables e.g.: `sscanf(argv[2],"%f",&float_value);`
(c.f. part 4 of last week's lab exercise)

- Change the *declaration* of `insert` to use a pointer parameter:
  `..., int *nextinsert)`
- Change the *call* of `insert` to use a pointer parameter:
  `..., &nextinsert)`
- Change each *use* of `nextinsert` inside `insert` to access the integer value via the pointer:
  `...*nextinsert...`
  You may need to use brackets e.g. `(*nextinsert)++`

Your program should now behave properly again.

Marking Process

You will continue using your program in lab exercise 5, so make sure you leave it in a good state.

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

They will look for: `arrays.c`

The marks are awarded as follows:

```
3 marks: Part 1 working completely
3 marks: Part 2 working completely
2 marks: Part 3 working completely
2 marks: Part 4 working completely

Total 10
```